A Cache-Efficient Iterative Sparse Triangular Solver

Speeding Up Solving Sparse Triangular Systems

AM3001 Bachelor Project

I.W.D.H. Rehorst

Delft University of Technology

TU Delft

A Cache-Efficient Iterative Sparse Triangular Solver

Speeding Up Solving Sparse Triangular Systems

by

I.W.D.H. Rehorst

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on **10/07/2025**

Thesis Committee:	Dr. J. Thies, TU Delft, Supervisor
	Dr. D.J.P. Lahaye, TU Delft
Project Duration:	March, 2025 - Juli, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover: DelftBlue Supercomputer by Sam Rentmeester (Modified)

TU Delft

Abstract

Sparse triangular solves (SpTRSV) form the latency–critical inner loop of many direct and iterative solvers, but strong data dependencies limit thread–level parallelism and make the kernel dominated by memory–latency. This thesis explores whether redundant computation can be exchanged for improved data locality to accelerate SpTRSV on cache-based multi-core CPUs.

The proposed two-phase algorithm first uses the Recursive Algebraic Colouring Engine (RACE) to permute an arbitrary sparse triangular factor into a block-lower–bidiagonal form whose diagonal blocks fit into a private L2 cache. Each block is then solved twice: an independent provisional pass followed by a lightweight correction that re-uses the still-resident data. The resulting task graph halves the critical path, exposes ample parallelism, and leaves total memory traffic unchanged.

An OpenMP5.0 implementation in `C++17` employs the new `affinity` clause so that producer–consumer task pairs are likely to run on the same core. Performance is compared against IntelMKL's sparse triangular routine and the Kokkos-Kernels `sptrsv`, while LIKWID counters validate cache behaviour.

On a 24-core Cascade Lake node of the DelftBlue supercomputer the task graph achieves a strong-scaling speed-up between 1.25 and 1.45 that saturates after roughly six threads; Kokkos attains similar absolute time only beyond sixteen threads. With 24 threads the solver is up to an order of magnitude faster than single-threaded MKL, and the `affinity` hint alone accelerates execution by 1.1–2×, confirming the importance of cache reuse. MKL remains preferable for small or highly structured SPD matrices, but on irregular, memory-bound factors the new solver rivals state-of-the-art libraries despite its simple prototype status.

Contents

1 Introduction

Solving triangular linear systems is an operation that appears in a broad spectrum of scientific and engineering applications. Sparse triangular systems emerge naturally from LU decompositions, from incomplete-factorization preconditioners, and within sub-domain solvers in domain-decomposition schemes; Consequently, they are fundamental to finding the numerical solution of many partial differential equations. Although forward and backward substitution needed to solve such a triangular system define a strictly sequential data dependency, the sparsity makes parallelization possible. Unfortunately, prevailing methods often buys this parallelism at the expense of regular data accesses (which leads to poor spatial cache locality) so the CPU ends up starved by cache misses or stalled by insufficient parallel work, leading to sub-optimal performance on modern hardware architectures.

In this report a method is introduced that tries to increase parallelism by introducing redundant computations, mitigating the effect of that redundancy by temporal cache locality.

The next section begins with formally defining the problem and its mathematical formulation in Section 1.1. Following this in Section 1.2, the goals of this project are given. Section 1.3 gives a brief overview of the material covered in this report.

1.1. Problem Formulation

The efficient numerical solution of sparse triangular linear systems is a fundamental problem in scientific computing, arising in a wide range of applications, including sparse direct solvers and preconditioners for iterative methods. Specifically, we are concerned with solving linear systems of the form:

$$Lx = b, \tag{1.1}$$

where $L \in \mathbb{R}^{n \times n}$ is a sparse lower triangular matrix, $b \in \mathbb{R}^n$ is a given right-hand side vector, and $x \in \mathbb{R}^n$ is the unknown solution vector to be computed. A matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is called sparse when the vast majority of its N^2 entries are exactly zero. This operation is commonly referred to as Sparse Triangular Solve (SpTRSV). Note that L could also be a sparse upper triangular matrix, and the method that will be implemented in this report also works for upper triangular matrices, but for the sake of simplicity L is assumed to be a lower triangular matrix throughout this report unless noted otherwise.

In the context of modern high-performance computing, the development of efficient parallel algorithms for SpTRSV has become increasingly important. As hardware architectures continue to evolve towards manycore and heterogeneous designs, exploiting parallelism at multiple levels is essential to achieve high throughput. However, the inherently sequential nature of triangular solves poses a significant challenge for parallel execution, limiting scalability and performance on such architectures. These challenges will be further discussed in the next section.

The primary objective of this work is to develop and analyse a SpTRSV method that is optimized specifically for modern multi-core processors with large caches. In particular, we are interested in approaches that leverage task-based parallelism and exploit the structure of the problem to expose

concurrency, while mitigating the negative impact of data dependencies and irregular memory access patterns. A full overview of goals for this project are given in Section 1.2.

It is important to emphasize that the focus lies on the solution of very large sparse linear systems, where the problem size is sufficiently large to amortize any algorithmic overhead introduced by parallelization strategies. Techniques such as structural reordering, redundant computations, and task scheduling inherently involve additional computations and management costs. For small problem sizes, these overheads may outweigh the performance gains. However, for large-scale systems, as encountered in applications like finite element simulations, computational fluid dynamics, or large-scale graph analytics, such methods become increasingly effective and necessary to fully utilize modern hardware capabilities. Throughout this report, we thus primarily consider scenarios where the size and sparsity of the matrix L justify the introduction of these techniques.

1.2. Objectives

The thesis is motivated by the broad ambition to answer: Can SpTRSV be sped up by using redundant computations to improve data locality?

From that question three concrete objectives are derived:

1. **Identify and exploit parallelism in SpTRSV**

   Show that a task- or block-based schedule lets many CPU cores advance simultaneously without data hazards. Any clear acceleration on multi-core hardware is considered a successful indication.

2. **Maximise cache reuse**

   Reorder computations so that data already loaded into cache is reused more frequently than in a naïve row-wise solve. Lower cache-miss activity or higher arithmetic intensity, as measured with hardware counters, would signal improvement.

3. **Quantify the benefit over existing methods**

   Benchmark the prototype against a well-known vendor libraries. The aim is to find out in which cases our approach is favourable.

1.3. Thesis Outline

The remainder of this report is organised as follows. Chapter 2 surveys the state of the art in sparse–triangular solves, tracing the evolution from thread-level parallelism to modern task-based, cache-aware strategies and exposing the performance gap this project targets, while also giving an introduction to the most important concepts needed to understand this thesis. Chapter 3 then shows the solution method used to solve sparse triangular linear systems introduced in this report. Building on this foundation, Chapter 4 details the implementation of our task-based block bi-diagonal solver. The quantitative impact of these design decisions is presented in Chapter 5, which compares our solver against other methods for SpTRSV across a suite of real-world matrices. Finally, Chapter 6 summarises the main findings, reflects on current limitations, and outlines avenues for future research.

2

Literature Review

This chapter presents a review of the literature related to efficiently solving sparse lower triangular systems. An overview of modern CPU architectures is then provided in Section 2.3 to highlight the importance of memory hierarchy and parallelization strategies in overcoming these challenges. Subsequently, the chapter surveys existing solution methods in Section 2.4, examining the evolution of algorithmic approaches, from classical techniques to highly optimized vendor libraries and task-parallel algorithms. Finally, in Section 2.5 the motivation for the research presented in this thesis is given.

2.1. Key Concepts

Before surveying existing work, we briefly recall the notions that occur throughout this report.

2.1.1. Sparse Matrix

A matrix is called sparse when only a small fraction of its N^2 entries is non-zero. Storing such matrices in compressed formats (e.g. compressed-row storage, CSR) avoids both the memory requirement and the memory traffic (i.e. the bytes that must travel between main memory and the CPU) associated with zero elements.

2.1.2. Cache Hierarchy and Locality

Modern CPUs bridge the $\approx 100\times$ latency gap between main memory and the core by staging data in one or more on-chip caches. Two complementary access patterns decide whether an algorithm makes good use of these caches [13]:

- **Spatial locality.** Data are moved in fixed quanta called *cache lines* (typically 64 B = 8 doubles). Accessing $x[i]$ therefore brings $x[i+1]$–$x[i+7]$ into the cache "for free", so linear or small-stride loops reuse bytes that are already resident. (Very large strides that are exact powers of two can map the same cache set and cause cache thrashing but this is rare in sparse solvers and not pursued further here.)
- **Temporal locality.** Once a datum is in cache, any access before it is evicted avoids a full round-trip to DRAM and is effectively "free" compared with the $\approx 100\times$ slower memory access. The task-based solver in this thesis is designed to maximise such short-lived re-uses, for instance by scheduling the two consecutive triangular solves that touch the same block on the same core.

2.1.3. Thread-level Parallelism (TLP)

On multi-core processors several hardware threads can execute simultaneously. Exploiting *TLP* means decomposing a computation into independent chunks that progress in parallel without violating data dependencies.

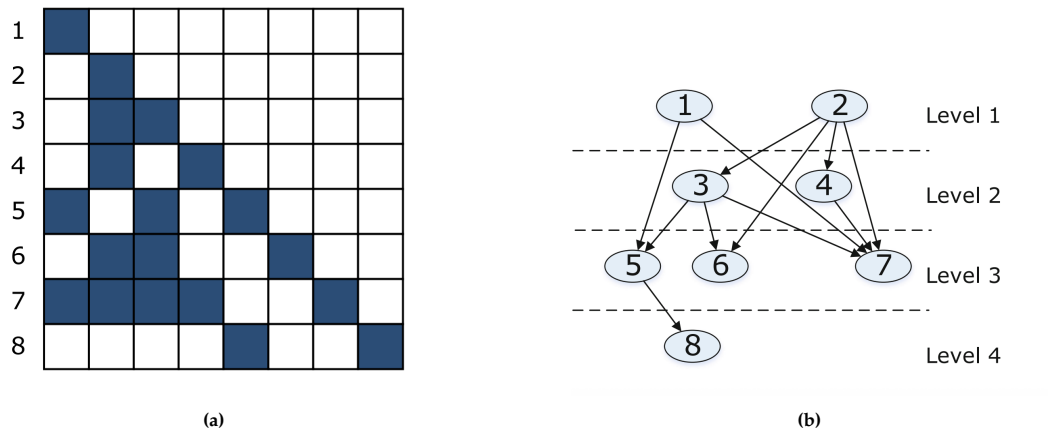


Figure 2.1: (a) Non-zero pattern of a lower triangular sparse matrix and (b) its corresponding task dependency graph of forward solver with level annotations [11]

2.2. Background and Challenges in Sparse Triangular Solvers

In this section, first a simple method for solving triangular linear systems is introduced, its sequential nature will be shown, as well as possible opportunities and challenges for parallelization of this method.

2.2.1. Forward Substitution

To solve a lower triangular linear system $Lx = b$ the standard method used is forward substitution. This procedure computes the unknowns sequentially from top to bottom, exploiting the triangular structure of the matrix. The i-th unknown is computed using:

$$x_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} x_j \right) \tag{2.1}$$

where it is assumed that $L_{ii} \neq 0$ for all i. Since each x_i depends on $x_1, \ldots, x_{i-1}$ the algorithm proceeds row-by-row, computing one unknown at a time.

Forward substitution is both simple and numerically stable for well-posed problems, and forms the computational core of many direct and iterative solvers in numerical linear algebra [8]. The major limitation of forward substitution is its strict data dependency: to compute x_i, all the dependent x_j terms (with $j < i$) must already be computed. This imposes a strong sequential execution order that hinders parallelization on modern hardware.

However, in the context of sparse matrices, not every entry below the diagonal is non-zero. Consequently, many of the dependencies in the summation are absent, and some x_i may only depend on a small subset of earlier unknowns. This sparsity opens the door to parallelism, as independent or loosely coupled rows can potentially be solved concurrently. The dependencies of the triangular system can be represented as a directed acyclic graph, where nodes represent unknowns and edges indicate dependencies. Analysing this graph reveals the concurrency that can be exploited in the solve phase. An example of this is given in Figure 2.1 [11].

2.2.2. Challenges Arising from Dependency Graphs

While the DAG view of a sparse triangular system enables the identification of parallelism, it also highlights several challenges. First, the depth of the dependency graph, which corresponds to the length of the critical path, ultimately limits the degree of parallelism that can be exploited [11].

Second, the irregularity of the sparsity pattern leads to non-uniform workloads across parallel threads. Some rows may become ready to compute sooner than others, causing load imbalance. Managing these dynamic dependencies efficiently requires intelligent scheduling strategies, often using techniques such as level scheduling, graph colouring, or block-based decomposition [8].

Third, traversing a sparse matrix produces irregular, non-contiguous memory accesses. Besides lowering cache hit rates, two threads may unknowingly touch different elements that reside in the same 64-byte cache line. If one of them writes, the line must be passed back and forth between their private caches (a phenomenon called false sharing), adding extra cache-level synchronisation even in the absence of true data conflicts [16].

Thus, while sparsity makes parallelism possible, achieving high performance in practice requires addressing these architectural and algorithmic challenges. The remainder of this thesis focuses on methods that mitigate these challenges.

2.3. Parallelization

The efficient solution of sparse triangular linear systems is not only a question of numerical methods, but also heavily depends on the characteristics of modern computer architectures. To understand the performance challenges and design choices in sparse triangular solvers, it is essential to consider how data is stored, moved, and processed at the hardware level. This section provides a brief overview of the relevant concepts in computer organization and parallel computing that are necessary to understand the techniques explored in this thesis.

2.3.1. Memory Hierarchy and Cache Efficiency

When solving large systems of equations numerically, a common misconception is that the computational speed of the processor (measured in FLOPs, or floating-point operations per second) is the main limiting factor. In reality, especially for sparse problems, the primary bottleneck often lies in memory access.

To mitigate this, processors use a hierarchy of caches (L1, L2, L3), which are small but fast memory units located closer to the CPU cores. Accessing data from L1 cache can be around 100× faster than accessing it from main memory [7]. Consequently, algorithms that reuse data in cache (temporal locality) and access data in a way that aligns with cache-friendly patterns (exploiting spatial locality) are significantly faster in practice.

Sparse triangular solvers, however, face a unique challenge: the matrix data is sparse, meaning it contains mostly zero entries, and is typically stored in a compressed format. This leads to indirect memory accesses through index arrays, causing irregular and often unpredictable access patterns. Such irregularity makes it difficult to exploit cache locality, resulting in frequent cache misses and wasted computational resources.

For these reasons, improving cache efficiency becomes more important than reducing the number of arithmetic operations. Techniques like blocking (dividing the matrix into small sub-matrices, or blocks, that fit into cache) and matrix reordering aim to improve data locality, reducing cache misses during the solve phase.

2.3.2. Instruction-Level and Thread-Level Parallelism

While individual CPU cores have become only marginally faster over the last decade, overall computational capacity has increased through parallelism. Modern processors feature multiple cores that can operate simultaneously, allowing multiple operations to proceed in parallel. Modern CPUs can exploit multiple forms of parallelism which are discussed here below.

Instruction-Level Parallelism (ILP): Within a single core, multiple instructions can be processed in parallel through pipelining and superscalar execution. However, ILP is largely automatic and limited by data dependencies; It does not suffice to accelerate algorithms like sparse triangular solves, where computations often depend on results from earlier steps [13].

Thread-Level Parallelism (TLP): This refers to running multiple threads of execution across multiple CPU cores. Threads can operate on independent data or different parts of a problem simultaneously. For instance, if two parts of a matrix can be processed independently, they can be assigned to different threads, leveraging the parallelism of the hardware [13].

2.3.3. Task Parallelism and Dependency Management

Sparse triangular solves require fine-grained and irregular parallelism. Here, the concept of task parallelism becomes useful.

In task-based parallelism, the problem is divided into discrete units of work called tasks. Tasks can be thought of as mini-programs that perform a specific computation (e.g., solving for a block of unknowns in a triangular solve). These tasks can be executed concurrently, as long as data dependencies are respected.

Modern parallel programming frameworks like OpenMP provide constructs to express such task dependencies explicitly. For instance, OpenMP 4.5 and later versions allow developers to annotate tasks with depend clauses, specifying which data is read and written by each task [2]. The runtime system then schedules tasks dynamically, ensuring correctness while trying to maximize concurrency.

In the context of sparse triangular solvers, we can express the solution process as a Directed Acyclic Graph (DAG), where nodes represent computational tasks (e.g., solving a block of unknowns) and edges represent data dependencies between these tasks. The DAG-based task scheduling enables the solver to overlap independent computations, making effective use of multiple threads. It also allows for redundant computations where beneficial, trading extra arithmetic for better cache reuse and parallel efficiency.

2.4. Existing Methods

Over the years, a variety of methods and optimizations have been proposed to address the issues mentioned in the previous sections. In this section, we provide an overview of the most relevant approaches and introduce the widely used libraries against which we will benchmark our own method.

2.4.1. Level Scheduling

One of the earliest and most widely adopted strategies for parallel sparse triangular solves is level scheduling. The idea is to preprocess the matrix to identify sets of unknowns that can be solved independently because they do not depend on each other. These sets are called levels. Solving proceeds level by level, with all unknowns within a level computed in parallel, followed by a global barrier that ensures every thread has finished that level before the next one can begin.

While straightforward, level scheduling is limited in its effectiveness because the number of available levels often grows logarithmically with the problem size, resulting in limited parallelism for large sparse systems. Moreover, extracting levels adds preprocessing overhead, and the benefit depends heavily on the sparsity structure of the matrix [8].

2.4.2. Blocked and Supernodal Methods

A different strand of work targets blocked or supernodal representations. During the factorisation step these approaches group contiguous columns whose non-zero patterns are identical (supernodes) so that the expensive updates can be cast as dense matrix–matrix multiplies and handled by highly-tuned BLAS Level-3 kernels. In the subsequent triangular solve phase each supernode is treated as a dense triangular block, which reduces indirect indexing and improves cache reuse, although the per-block operation is now a matrix–vector multiply (BLAS Level-2) and therefore still memory-bound.

Supernodal techniques pay off when the matrix contains large, regular blocks. For matrices that remain highly irregular the gains taper off, and additional fine-grained parallel strategies become necessary [18].

2.4.3. Dependency Graph Approaches and Task Parallelism

More recently, methods based on dependency graphs and task-based parallelism have gained prominence. Instead of grouping unknowns into coarse-grained levels, these approaches represent the triangular solve as a Directed Acyclic Graph (DAG) of fine-grained tasks, with edges representing data dependencies [8].

Frameworks like OpenMP tasking and Kokkos task DAGs allow expressing these dependencies explicitly. The runtime system dynamically schedules tasks while respecting data dependencies, enabling finer granularity of parallelism and better utilization of modern multicore processors.

These DAG-based approaches can adapt to highly irregular sparsity patterns and allow advanced strategies like redundant computations and asynchronous execution, which can improve both cache locality and load balancing across cores.

2.4.4. Specialized Sparse Triangular Solvers in Libraries

Several mature HPC libraries ship hand-tuned sparse-triangular kernels. On CPUs the best-known package is the Intel Math Kernel Library (Intel MKL). Its sparse BLAS call `mkl_sparse_d_trsv` is aggressively vectorised and cache-aware, yet the actual forward or backward substitution still executes on a single OpenMP thread. Because the code base is highly optimised, this serial kernel often rivals or exceeds naïve threaded versions and therefore remains a good baseline for CPU studies [3]. In the present work MKL serves as a reference for what can be achieved with one core and near-optimal data locality.

To place the new solver in a genuinely parallel context the experiments also use KokkosKernels, the task-DAG backend of the Kokkos ecosystem. Its `sptrsv_symbolic/solve` pair offers several level-scheduling algorithms that exploit nested parallelism on multicore CPUs and GPUs while remaining fully open source and architecture agnostic. Kokkos therefore represents the current state of portable, task-parallel techniques aimed at shared-memory machines.

Packages such as cuSPARSE for NVIDIA GPUs, the Ifpack2/Tpetra modules inside Trilinos, or the classical Hypre library also provide SpTRSV routines, but they are not benchmarked here. The proposed algorithm is tailored to cache-based shared-memory CPUs, and Kokkos already embodies the modern parallel strategies that those other libraries increasingly adopt.

The performance study thus compares three solvers: the serial but highly tuned MKL routine, the fully parallel implementation from KokkosKernels, and the method introduced in this thesis. In Chapter 4 these libraries will be discussed in greater detail.

2.5. Motivation for Our Approach

While existing SpTRSV methods and libraries offer a range of techniques to address the challenges posed by sparsity and parallelism, they often involve trade-offs between generality, scalability, and memory efficiency. Level scheduling is constrained by limited parallelism, supernodal methods require favourable matrix structures to be effective, and task-based runtimes still face difficulty in balancing overhead with computational benefit, especially for fine-grained tasks. Moreover, many existing solvers are designed to perform optimally on matrices with specific sparsity patterns or hardware characteristics, limiting their general applicability. In contrast, the method proposed in this thesis is motivated by the need for a sparse triangular solver that is both cache-efficient and well-suited to shared-memory parallel architectures, especially in the context of very large systems where such considerations have a measurable impact on performance. By using redundant computation, block decomposition, and task parallelism, the proposed approach seeks to exploit both structural and hardware-level opportunities for concurrency that are often underutilized in conventional solvers.

3

Methodology

This chapter develops the complete solution strategy step by step. Section 3.1 begins by transforming the raw triangular factor with an ordering that exposes an explicit block structure. Once this structure is in place, Section 3.2 reviews the classical sequential forward–substitution on the resulting block–bidiagonal matrix, establishing a performance baseline. Section 3.3 introduces the method proposed in this thesis for the 2×2 case. Section 3.4 generalises the same ideas to a chain of k blocks and discusses the scheduling rules that guarantee correctness and parallel efficiency. Finally, Section 3.5 analyses the computational and memory costs of the proposed method.

3.1. Problem Simplification and Reordering

We begin by recalling the problem that was stated in Section 1.1: solving a sparse lower triangular system of the form

$$Lx = b, \tag{3.1}$$

where $L \in \mathbb{R}^{n \times n}$ is a sparse lower triangular matrix, $b \in \mathbb{R}^n$ a known right-hand side vector, and $x \in \mathbb{R}^n$ the solution vector to be computed.

Although simple in theory, the structure of L imposes a strict partial ordering on the computation of the unknowns x_i, as each value x_i can only be computed once all its dependencies have been resolved, as was shown in Section 2.2. This sequential nature limits the scalability of classical forward substitution methods on parallel hardware. To overcome this, we seek to expose and exploit parallelism by reordering the matrix in a way that groups independent computations into blocks.

3.1.1. Matrix Reordering with RACE

To enable parallelism, a reordering of the matrix L is performed using the Recursive Algebraic Colouring Engine (RACE) library [1]. RACE first builds the row–dependency graph $G(V, E)$ from the sparsity pattern of L ($i \to j \Leftrightarrow x_j$ depends on x_i) and then performs a parallel breadth-first search starting from row 0. Every node reached at distance ℓ from the seed is assigned to level ℓ. Hence a row in level k can only depend on rows in levels k or $k-1$ (for a lower-triangular L). Dependencies within a level are allowed and give rise to the triangular blocks on the diagonal in what follows.

From this colouring a permutation matrix $P \in \mathbb{R}^{n \times n}$ is extracted, which is used to permute the original system $Lx = b$. The reordered matrix is given by

$$\tilde{L} = PLP^T. \tag{3.2}$$

Here L_i collects all rows of level i and remains lower triangular, while B_i contains the non-zeros that link level i to the previous level $i-1$. Because no row touches levels further away, the matrix is

block-lower-bidiagonal. The general structure of the reordered matrix $\tilde{L}$ is given below:

$$\tilde{L} = \begin{bmatrix} L_1 & 0 & \cdots & 0 & 0 \\ B_2 & L_2 & & & \\ 0 & B_3 & L_3 & & \\ \vdots & 0 & \ddots & \ddots & 0 \\ 0 & & & B_k & L_k \end{bmatrix} \tag{3.3}$$

where each L_i is a square sparse lower triangular block, and B_i is a sparse rectangular block capturing dependencies between level $i-1$ and level i. Importantly, the blocks L_i are now independent of one another and can be solved in parallel, subject only to their dependencies through the B_i blocks.

RACE can recursively split large levels or merge thin ones, which lets us tune the final block size almost independently of the original sparsity pattern. In practice we choose the threshold so that the number of blocks matches (or exceeds) the available hardware threads while keeping each L_k small enough to fit into the private cache of a core.

The remainder of this chapter develops an efficient task-based algorithm for solving the resulting block-lower-bidiagonal system.

3.2. Solving the Block-Bidiagonal System

After reordering with RACE the system $Lx = b$ is transformed into the block-lower-bidiagonal form:

$$\begin{bmatrix} L_1 & 0 & \cdots & 0 & 0 \\ B_2 & L_2 & & & \\ 0 & B_3 & L_3 & & \\ \vdots & 0 & \ddots & \ddots & 0 \\ 0 & & & B_k & L_k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ v_k \end{bmatrix} \tag{3.4}$$

It is very important to note here that the individual entries x_i of $x \in \mathbb{R}^n$ and b_i of $b \in \mathbb{R}^n$ are not scalars, but are vectors with a size that corresponds to the dimension of the corresponding block L_i.

The linear system given by equation 3.4 can be solved by a block forward substitution:

$$x_1 = L_1^{-1} b_1 \tag{3.5}$$

$$x_i = L_i^{-1}(b_i - B_i x_{i-1}) \quad i \in \{2, 3, \ldots, k\} \tag{3.6}$$

Although the block forward substitution is simple and robust, it is strictly sequential: one must finish x_1 before starting x_2 and so on, leaving no exploitable parallelism for a single right-hand side. In addition, each triangular block L_i is fetched from main memory, applied exactly once, and then evicted, so the memory traffic is dominated by compulsory loads.

Because of the limitations given above, in the next section, a redundant two-block strategy that overlaps the factor applications of neighbouring blocks and reuses L_i while it is still resident in cache. First a 2×2 prototype is given, and then the idea is generalized to the complete block-bidiagonal matrix.

3.3. The 2×2 Case

To illustrate how redundancy can unlock both parallelism and cache reuse, first the smallest non-trivial instance of the block-bidiagonal system is considered, namely $k = 2$. After the RACE permutation the linear system reads:

$$\begin{bmatrix} L_1 & 0 \\ B_2 & L_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \qquad L_i \in \mathbb{R}^{m_i \times m_i},\ x_i, b_i \in \mathbb{R}^{m_i} \tag{3.7}$$

A direct block forward substitution proceeds

$$x_1 = L_1^{-1} b_1, \tag{3.8}$$

$$x_2 = L_2^{-1}\big(b_2 - B_2 x_1\big). \tag{3.9}$$

The two solves form a strict chain, at most one diagonal block can be processed at a time, and each triangular factor is read from memory only once but is also used only once, resulting in poor cache utilisation.

We trade redundant work for overlap and locality:

$$\textbf{Provisional step (parallel)} \qquad \hat{x}_1 = L_1^{-1} b_1, \qquad \hat{x}_2 = L_2^{-1} b_2, \tag{3.10}$$

$$\textbf{Correction step} \qquad x_1 = \hat{x}_1, \qquad x_2 = L_2^{-1}\big(b_2 - B_2 \hat{x}_1\big). \tag{3.11}$$

The two provisional solves are independent and can run concurrently. The correction then reuses the already-fetched factor L_2 while it is still warm in the private L2 cache of the core that computed it, avoiding a second trip to main memory. The critical-path length (often called the span) drops from two full triangular solves to $\frac{3}{2}$, at the cost of only one redundant application of L_2.

3.3.1. Key Observations

The redundancy incurs one extra triangular solve with L_2, however, it shortens the critical path because the first two solves overlap fully, leaving only a single dependent step. At the same time the cache behaviour improves, since the factor L_2 is loaded from memory only once yet applied twice while it remains hot.

The same "provisional + correction" idea can be applied recursively to the k-block bidiagonal system, yielding a pipeline in which every diagonal block L_i is reused before it leaves the cache, while task dependencies preserve correctness.

3.4. Extension to a General k-block System

Consider the reordered system containing $k \geq 2$ diagonal blocks

$$\begin{bmatrix} L_1 & & & & \\ B_2 & L_2 & & & \\ & B_3 & L_3 & & \\ & & \ddots & \ddots & \\ & & & B_k & L_k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \end{bmatrix}, \qquad L_i \in \mathbb{R}^{m_i \times m_i},\ B_i \in \mathbb{R}^{m_i \times m_{i-1}}. \tag{3.12}$$

In the previous section it was shown how this system can be solved for the $k = 2$, now this method will be extended for a $k \geq 2$ system.

3.4.1. Two-Phase Redundant Strategy

The overlap idea from the 2×2 prototype is generalised by splitting the work into two phases.

Phase 1 (provisional solves): Each diagonal block L_i is solved independently, so up to $\min(k, p)$ blocks can execute in parallel on a machine with p hardware threads. If $k > p$ the surplus blocks are queued as additional tasks:

$$\hat{x}_i \;=\; L_i^{-1} b_i, \qquad i = 1, \ldots, k, \tag{3.13}$$

ignoring inter-level couplings. All k solves run concurrently.

Phase 2 (correction wave): A single forward sweep injects the missing couplings:

$$\begin{aligned} x_1 &= \hat{x}_1, \\ x_i &= L_i^{-1}\big(b_i - B_i x_{i-1}\big), \qquad i = 2, \ldots, k. \end{aligned} \tag{3.14}$$

Each L_i is therefore applied twice, but the interval between applications is only one neighbouring block, so the factor can remain in the private L2 cache. The application of these phases can essentially be seen as separate tasks that are applied to solve equation 3.12. A task dependency schedule is what follows and will be shown in the next section.

3.4.2. Task Schedule

Each block L_i is processed twice: a provisional triangular solve that ignores the sub-diagonal block B_i, followed by a correction that injects the missing contribution $B_i x_{i-1}$. These two steps can be labelled as:

$$T_i^{(1)} \text{ (provisional)}, \qquad T_i^{(2)} \text{ (correction)}, \qquad i = 1, \ldots, k.$$

For every block i the right-hand side of the provisional system is

$$b_i^{(1)} = b_i,$$

which depends on the original right-hand side only. Consequently no data produced by any other block are required and the tasks $T_1^{(1)}, \ldots, T_k^{(1)}$ are mutually independent. The correction task must re-use the provisional solution of the same block:

$$T_i^{(1)} \longrightarrow T_i^{(2)}, \qquad i = 1, \ldots, k.$$

This edge reflects the need to keep L_i and $x_i^{(1)}$ in cache so that the second application of the factor is inexpensive in terms of memory traffic.

Before block i can apply its correction it needs the fully corrected solution of the $(i-1)$-th block, because the residual vector is

$$b_i^{(2)} = b_i - B_i x_{i-1}^{(2)}.$$

Thus the corrections are linked by a forward chain:

$$T_{i-1}^{(2)} \longrightarrow T_i^{(2)}, \qquad i = 2, \ldots, k.$$

After the first two levels have completed their provisional solves the pipeline is full:

time step	1	2	3	...	k
concurrent tasks	$T_1^{(1)}$	$T_2^{(1)}, T_1^{(2)}$	$T_3^{(1)}, T_2^{(2)}$	...	$T_k^{(1)}, T_{k-1}^{(2)}$

With p hardware threads available, all $T_1^{(1)}, \ldots, T_k^{(1)}$ can, in principle, start concurrently. As soon as the first provisional result $\hat{x}_1$ is ready the correction chain $T_1^{(2)} \to T_2^{(2)} \to \cdots$ begins and runs in a pipelined fashion. During most of the execution time two kinds of work overlap:

$$\underbrace{T_i^{(1)}}_{\text{full triangular solve}} \quad \| \quad \underbrace{T_{i-1}^{(2)}}_{\text{cheap correction using cached } L_{i-1}}$$

Once all provisional solves have completed, the remaining $T^{(2)}$-pipeline is bandwidth-bound but substantially cheaper than a full triangular solve because it reuses the already-resident factor L_i. Consequently the critical path is no longer the full sequential chain of k triangular solves but consists of one parallel step for the $T^{(1)}$ tasks, plus roughly $(k-1)$ lightweight corrections, which is shorter and much better parallelised than the baseline algorithm.

In practice the number of hardware threads, p, is fixed by the node architecture, whereas the number of logical tasks equals $2k$. We therefore let the OpenMP task scheduler assign work dynamically. A comprehensive discussion of the concrete OpenMP implementation is given in Chapter 4.

In the next section, the cost of this method will be analysed.

3.5. Cost Analysis of the Proposed Solver

This section estimates the arithmetic work, data-movement volume, and parallel scalability of the two-block overlapping algorithm from Section 3.2. All quantities are expressed in terms of the non-zero counts of the block–bidiagonal matrix

Let the reordered system be partitioned into k diagonal blocks $L_i \in \mathbb{R}^{m_i \times m_i}$ and sub-diagonal coupling blocks $B_i \in \mathbb{R}^{m_i \times m_{i-1}}$ as discussed in Section 3.1. Denote their non-zero counts by

$$\ell_i = \mathrm{nnz}(L_i), \qquad \beta_i = \mathrm{nnz}(B_i), \qquad i = 1, \ldots, k,$$

and define

$$\ell = \sum_{i=1}^{k} \ell_i, \qquad \beta = \sum_{i=2}^{k} \beta_i.$$

3.5.1. Baseline Cost

With plain block forward substitution the i-th step performs

$$\begin{aligned} \mathrm{flop}(i) &= 2\,\ell_i \quad \text{(triangular solve)} && (3.15)\\ &\quad + 2\,\beta_i \quad \text{(update } b_i - B_i x_{i-1}\text{)}, && (3.16) \end{aligned}$$

so that the total work is

$$W_{\text{base}} = 2(\ell + \beta). \tag{3.17}$$

Each non-zero of L_i or B_i is streamed once from main memory, so the matrix traffic is

$$Q_{\text{base}} = (\ell + \beta)\,\texttt{sizeof(float64)}.$$

The vectors b and x add only $O(\sum_i m_i)$ bytes, which is negligible for the large, highly sparse matrices considered here where $\ell + \beta \gg \sum_i m_i$. In practice b is read exactly once and x is written once.

Because every block depends on the fully computed result of its predecessor, the algorithm is strictly sequential: the critical path consists of the entire chain of k block solves and therefore admits no speed-up from parallel execution.

3.5.2. Overlapping Two-Block Algorithm

The overlapping schedule executes two triangular solves per diagonal block:

$$W_{\text{overlap}} = 4\ell + 2\beta = 2\,W_{\text{base}} - 2\beta. \tag{3.18}$$

Thus the floating-point cost on the L_i doubles, whereas the work on the B_i is unchanged.

Because the provisional and correction solves of the same block are consecutive on the same thread (Section 3.4.2), the factor L_i is kept in cache and does not incur a second main-memory load:

$$Q_{\text{overlap}} = (\ell + \beta)\,\mathrm{sizeof(float64)} = Q_{\text{base}}. \tag{3.19}$$

Consequently the algorithm trades pure compute for a net reduction in bytes per flop and is therefore more compute-intensive.

With the pipeline full, one provisional task $T_{i+1}^{(1)}$ overlaps with one correction task $T_i^{(2)}$. The length of the dependency chain (hence the critical path) shrinks to

$$\mathrm{depth}_{\text{overlap}} = \left\lceil \frac{k+1}{2} \right\rceil,$$

yielding a theoretical speed-up factor of approximately 2 over the baseline for large k.

The overlap pattern implies that exactly two dependent tasks are ready at every step of the pipeline:

1. the current correction task $T_i^{(2)}$
2. the next provisional task $T_{i+1}^{(1)}$.

Therefore, as soon as $p \geq 2$ hardware threads are available, the span term S, where S is thus the length of the critical path, can be executed at full speed and adding further threads cannot shorten S any more.

Although the critical path is saturated with two threads, a larger thread pool improves performance through the work term. All provisional tasks $T_j^{(1)}$ for $j > i + 1$ are already free of dependencies and can be scheduled eagerly. With p threads the upper bound for the runtime becomes

$$T_p \;\leq\; \frac{W}{p} \;+\; S \quad\Longrightarrow\quad \text{speed-up}(p) \;=\; \frac{W}{W/p + S}.$$

For $p = 2$ the span is saturated and the speed-up is $\approx \frac{W}{W/2+W/2} = 1$ (span-limited). For $2 < p \ll k$ the work term $\frac{W}{p}$ keeps shrinking, so throughput keeps increasing until either memory bandwidth or task scheduling overhead dominates. In the idealised case $p \geq k$ every block obtains a private thread and the runtime approaches $T_p \approx \max_i (L_i)$, i.e. limited only by the largest individual block.

The overlapping two-block strategy doubles the arithmetic on the diagonal factors, but does not increase memory traffic and halves the span of dependent work. On modern cache-rich CPUs this trade-off leads to higher arithmetic intensity and, as demonstrated in Chapter 5, observable speed-ups over other sparse triangular solvers.

3.5.3. Pre–processing Overhead Due to RACE Reordering

Before any triangular solve can benefit from the overlapping schedule, the original sparse matrix L is permuted into block–bidiagonal form by the RACE library [1]. The pre–processing consists of three algorithmic stages whose costs are summarised below. Throughout we write

$$n = \text{rows}(L), \qquad \text{nnz} = \text{nnz}(L).$$

RACE requires a symmetric graph. Given that L is a lower triangular matrix, and is thus not symmetric, the pattern must first be augmented with its transpose. Let

$$\gamma \;=\; \frac{\text{nnz}(L \,\cup\, L^{\mathsf{T}})}{\text{nnz}} \quad (\gamma \geq 1)$$

quantify this overhead. Because it is assumed that L is a lower triangular matrix, it follows that $\gamma \leq 2$.

From the symmetrised pattern an n-vertex graph is built in linear time

$$T_{\text{graph}} = O\big(\gamma\,\text{nnz}\big), \qquad Q_{\text{graph}} = O\big(\gamma\,\text{nnz}\big) \text{ bytes}.$$

RACE applies a parallel colouring on sub-graphs level by level. The work is again linear in the edge set [1]:

$$T_{\text{colour}} = O\big(\gamma\,\text{nnz}\big), \qquad S_{\text{colour}} = O\big(\log n\big).$$

Finally the permutation vectors are formed and applied once to the matrix as well as to the right-hand side:

$$T_{\text{perm}} = O\big(\gamma\,\text{nnz}\big), \qquad Q_{\text{perm}} = O\big(\gamma\,\text{nnz}\big) \text{ bytes}.$$

The lower-triangular part of each diagonal block is then copied into a fresh CSR array, so every block occupies one contiguous memory segment. The copy is included in the T_{perm} term above.

Collecting the terms, the one-off pre-processing overhead is

$$T_{\text{RACE}} = T_{\text{graph}} + T_{\text{colour}} + T_{\text{perm}} = \Theta(\gamma\,\text{nnz}), \tag{3.20}$$

$$Q_{\text{RACE}} = Q_{\text{graph}} + Q_{\text{perm}} = \Theta(\gamma\,\text{nnz})\ \text{bytes}, \tag{3.21}$$

$$S_{\text{RACE}} = S_{\text{colour}} = O(\log n). \tag{3.22}$$

Thus the runtime overhead grows linearly with the size of the input matrix, and its critical path is negligible compared to the span of a single triangular solve.

In many applications (e.g. time stepping, Newton iterations) the same triangular factor is solved for numerous right-hand sides $b^{(j)}$. Let N_{rhs} be that number. Inserting the results of Sections 3.5.1–3.5.2, the amortised runtime per solve on p threads is bounded by

$$T_p^{\text{amort}} \le \frac{\gamma\,\text{nnz}}{N_{\text{rhs}}\,p} + \frac{W_{\text{overlap}}}{p} + S.$$

Hence for moderate N_{rhs} the RACE overhead becomes insignificant, whereas the benefits of the overlapping solver remain for every subsequent right-hand side.

3.5.4. Cost Overview

The cost model above highlights a deliberate trade–off:

1. *Extra arithmetic.* Doubling the work on the diagonal blocks adds 2ℓ flops, but these operations are both cache–resident and vector–friendly, so their cost on modern CPUs is low.
2. *Unchanged data volume.* Because every factor L_i is re-used while still in cache, the traffic $Q_{\text{overlap}} = Q_{\text{base}}$ remains memory–bound to the same degree as the traditional algorithm.
3. *Shortened span.* The overlapping schedule cuts the critical path from k to $\lceil (k+1)/2 \rceil$ triangular solves, enabling a theoretical $\approx 2\times$ speed-up at the task level and exposing ample work parallelism for any $p \ge 2$ threads.

When the one-off RACE permutation is amortised over multiple right-hand sides, the dominant costs are therefore

$$T_p^{\text{amort}} \approx \frac{4\ell + 2\beta}{p} + O(\log k), \qquad Q = (\ell + \beta)\,\text{sizeof(float64)},$$

so performance is ultimately limited by memory bandwidth once the span is saturated.

Chapter 4 turns these analytical insights into a concrete OpenMP task implementation, discusses practical issues, and details how LIKWID counters are used to validate the predicted data traffic. Experimental results, a comparison with Intel MKL and Kokkos and other baselines follow in Chapter 5, where the model derived here is confronted with real hardware measurements.

4

Implementation

This chapter translates the algorithmic ideas from Chapter 3 into an efficient, measurable software prototype. After introducing the sparse data structures (4.1) and preprocessing pipeline (4.2) we detail the task– based OpenMP implementation of the solver (4.3). The implementation of the reference methods given by Intel MKL (4.5) and Kokkos (4.6) will be discussed after that. The final sections discuss build integration (4.8) and the LIKWID–based performance instrumentation (4.7). Hardware platforms, compiler flags, and test matrices are collected later in Chapter 5.

4.1. Sparse Storage and Matrix Preparation

In order to efficiently solve sparse matrices an efficient storage method must be used, the compressed-row format has been chosen as mentioned previously, a small overview of this format is given below.

4.1.1. Compressed Sparse Row (CSR)

Every sparse $n \times n$ matrix is stored in classical CSR [15]:

$$\underbrace{\texttt{rowPtr}}_{n+1},\ \underbrace{\texttt{col}}_{\text{nnz}},\ \underbrace{\texttt{val}}_{\text{nnz}},$$

where

$$\text{rowPtr}[i] = \text{offset of first non-zero in row } i, \tag{4.1}$$
$$\text{col}[p] = \text{column index of non-zero } p, \tag{4.2}$$
$$\text{val}[p] = \text{value of non-zero } p. \tag{4.3}$$

The three arrays are allocated once and reused throughout the pipeline so that no format conversions occur after load time.

4.1.2. Matrix Ingestion

Matrices are read from `.mtx` files via a streaming parser that

1. collects triplets (i, j, a_{ij}),
2. sorts by (i, j) to build CSR in one pass,
3. discards explicit zeros.

An auxiliary routine `sparsemat::extract_triangle(bool lower)` keeps either the strictly lower or strictly upper part.

4.1.3. Synthetic Fill-In (Densification)

In practice one rarely solves a triangular system with the raw sparsity pattern of the coefficient matrix A. A numerical factorisation (e.g. $A = LU$ or $A = RR^{\mathsf{T}}$) introduces fill-in, so each factor is much denser than A itself. To mimic this situation without running an actual factoriser we densify the test matrices offline by replacing A with its third power A^3, which adds many structural non-zeros.

The helper routine

`sparsemat::multiply(const sparsemat&)`

implements a simple triple loop over rows, columns, and intersection lists. Because this densification is executed once during data preparation and never inside the timed kernels, its cost does not affect the performance results presented in Chapter 5.

4.2. Pre-Processing by RACE

The permutation step follows the theory of Section 3.1:

1. Build an undirected graph from `rowPtr/col`.
2. Invoke `RACE::colourGraph(...)` to obtain the level colouring.
3. Derive permutation vectors P, P^{-1} and the stage pointer `stagePtr`$[0 \dots k]$.

CSR is reordered in-place:

$$\mathrm{val}[p] \leftarrow a_{P^{-1}(i)\,P^{-1}(j)}, \qquad \mathrm{col}[p] \leftarrow P(j).$$

Finally the upper-triangular half is discarded and only the strictly lower part is kept,

$$\tilde{L} \;=\; \mathrm{tril}\big(P\,A\,P^{\mathsf{T}}\big),$$

which is stored as one contiguous CSR array, and this is the block-bidiagonal L factor that all subsequent kernels operate on.

4.3. Task-Based OpenMP Kernel

The theoretical schedule of Section 3.4.2 is realised with the OpenMP 4.5 task–dependency mechanism [2]. Listing 4.1 shows a condensed skeleton and the full routine appears in Appendix A.

Listing 4.1: Skeleton of the `blockBiDiagSolveTasks` kernel.

```
#pragma omp parallel default(none) shared(B,stagePtr,bp,xp,k)
{
  /* only one thread spawns the tasks */
  #pragma omp single
  {
    /* ---- Phase 1: provisional solves ---- */
    for (int i=0;i<k;++i) {
      int r0=stagePtr[i],  m=stagePtr[i+1]-r0;
      #pragma omp task                                    \
              depend(out: xp[r0:m])                       \
              firstprivate(r0,m)
      provisional_solve(i);
    }

    /* ---- Phase 2: correction solves ---- */
    for (int i=1;i<k;++i) {
      int r0 = stagePtr[i]      ,  m = stagePtr[i+1]-r0;
      int r00= stagePtr[i-1]    ,  m0= stagePtr[i]-r00;
      #pragma omp task                                    \
              depend(in:    xp[r00:m0])                   \
              depend(inout: xp[r0 :m ])                   \
              firstprivate(r0,m,r00,m0)
      correction_solve(i);
    }
    /* implicit taskwait */
```

```
26   }
27 }
```

The outer `#pragma omp parallel` creates a permanent thread team that persists for the whole solve. All shared objects (`B, stagePtr, bp, xp, k`) are therefore visible to every thread, avoiding repeated initialisation.

Exactly one thread enters the `single` region and enqueues all tasks. Every other thread waits for work in the OpenMP runtime's task queue.

The `depend` directives translate the logical edges

$$T_i^{(1)} \;\rightarrow\; T_i^{(2)}, \qquad T_{i-1}^{(2)} \;\rightarrow\; T_i^{(2)}$$

into concrete memory-region dependences:

- `out: xp[r0:m]`: the provisional task writes the slice $x_{r_0:r_1-1}$.
- `inout: xp[r0:m]`: the correction task both reads and writes the same slice, enforcing the self–dependence.
- `in: xp[r00:m0]`: a read-only dependence on block $i-1$ realises the pipeline constraint $T_{i-1}^{(2)} \rightarrow T_i^{(2)}$.

OpenMP guarantees that two tasks whose covered memory regions overlap cannot execute concurrently. Because the correction task's `inout` region is identical to its predecessor's `out` region, most runtimes schedule the pair on the same worker thread, thereby preserving cache residency of L_i and x_i. Formally the standard guarantees mutual exclusion when using the depend clause but it does not prescribe which worker thread executes either task. Most modern runtimes employ work–stealing queues that favour child–first scheduling [14]: a thread that finishes a task immediately proceeds with a dependent child from its local deque before attempting to steal from others. Empirically this heuristic places $T_i^{(2)}$ on the same core that just produced $T_i^{(1)}$, so both the factor L_i and the provisional solution slice x_i remain in the private cache hierarchy when they are reused, as was assumed in 3.4.2. However, the behaviour is an optimisation choice, not a contractual obligation of the OpenMP specification. To ensure that OpenMP actually handles the tasks as expected, the performance and memory usage will be measured using the LIKWID library. In the next section we will discuss an extra OpenMP directive that improves cache reuse.

All loop-invariant variables are `shared`. Block-local indices (`r0,m`) are passed `firstprivate`, which copies their value into the task's context to avoid false sharing.

The two *for* loops spawn a total of $2k-1$ tasks whose dependencies replicate exactly the graph of Section 3.4.2. After the start-up latency of the first block, the pipeline contains one correction task $T_i^{(2)}$ plus as many provisional tasks $T_j^{(1)}$ $(j > i)$ as the thread pool can accommodate, so the sequential correction chain overlaps with the independent provisional tasks. OpenMP's dynamic queue ensures that additional provisional tasks are pulled forward whenever idle threads exist, so the implementation can exploit thread counts $p > 2$ even though the span is already saturated with two.

4.4. Task-Based OpenMP Kernel with Affinity

Early experiments used exactly the schedule of Section 3.4.2 but without any affinity directives. The gcc runtime then relied on its default work–stealing policy: after a thread finished a provisional task it often, but not always, executed the dependent correction task. occasionally the task was stolen by a different core. The memory traces in Section 5.4.4 show the consequence. Across all matrices the private L2 caches delivered barely 10% of the required data, while the shared LLC registered gigabytes of traffic and a miss ratio below 9%, thus the provisional slice left the L2 before the correction step could reuse it.

To enforce cache reuse the implementation now adopts the affinity clause, introduced only in OpenMP 5.0 [9]. Listing 4.2 shows the relevant lines. Each provisional task specifies its output slice as `affinity(xp[r0])` and the matching correction task repeats the same anchor. The runtime regards this as a strong hint that both tasks should run close to the physical storage location of that

address, which in practice means the same core when the data reside in its private cache. Because affinity is a hint rather than a hard constraint the standard still allows the runtime to place tasks differently.

Listing 4.2: Core of `blockBiDiagSolveTasksAffinity`.

```
#pragma omp parallel default(none) shared(B,stagePtr,bp,xp,k)
{
  #pragma omp single
  {
    /* ---- Phase 1 : provisional solves ---- */
    for(int i=0;i<k;++i){
      int r0 = stagePtr[i];
      int m  = stagePtr[i+1]-r0;
      #pragma omp task                                    \
              depend(out: xp[r0:m])                       \
              affinity(  xp[r0]   )                       \
              firstprivate(r0,m)
      provisional_solve(i);
    }

    /* ---- Phase 2 : correction solves ---- */
    for(int i=1;i<k;++i){
      int r0  = stagePtr[i]      ,  m  = stagePtr[i+1]-r0;
      int r00 = stagePtr[i-1]    ,  m0 = stagePtr[i]-r00;
      #pragma omp task                                    \
              depend(in:    xp[r00:m0])                   \
              depend(inout: xp[r0 :m ])                   \
              affinity(  xp[r0]   )                       \
              firstprivate(r0,m,r00,m0)
      correction_solve(i);
    }
    /* implicit taskwait */
  }
}
```

All other implementation details remain unchanged: global data are shared, block-local indices are firstprivate, and the two loops spawn $2k - 1$ tasks whose dependencies reproduce the pipeline graph exactly.

4.5. Reference Implementation with Intel MKL

To put the task–based solver into context we benchmark it against Intel's highly optimised sparse triangular routine and use the latter as a truth model for numerical validation.

Intel oneMKL exposes two triangular kernels for sparse matrices stored in CSR format [6]:

1. `mkl_dcsrtrsv` — a single–call BLAS TRSV analogue that performs symbolic analysis and numerical solve in one go;
2. `inspector_exec` — the Inspector–Executor interface that separates pattern analysis (`mkl_sparse_optimize`) from the repeated numerical solves (`mkl_sparse_trsv`).

Because the matrices are solved for dozens of right–hand sides, the Inspector–Executor (IE) variant is chosen: its one–off symbolic phase is amortised just like the RACE permutation. According to the Intel MKL Developer Reference [6], all Level-3 BLAS and all Sparse BLAS routines except Level-2 sparse triangular solvers are threaded, the sparse triangular solve (`sptrsv`) kernel is thus inherently single-threaded. This is important to note when comparing our method to MKL.

For fair comparison the following wall-clock protocol is adopted:

1. allocate and initialise the right–hand side b and solution vector x once,
2. call the IE inspection phase outside the timed section,
3. repeat the numerical solve N_{rep} times,
4. record the median time t_{MKL}.

The same repetition / cache–flush procedure is applied to the task solver introduced in this report, the reported speed-up is therefore

$$\text{speed-up} = \frac{t_{\text{MKL}}}{t_{\text{tasks}}}.$$

After each run the residual

$$r \;=\; \| b - Lx \|_2$$

is computed in double precision and compared to the residual produced by MKL. The solutions are deemed equivalent if

$$\frac{|r_{\text{tasks}} - r_{\text{MKL}}|}{r_{\text{MKL}}} < 10^{-12}.$$

This is done to ensure the correctness of the solution found by the implemented solver.

The full function implementation can be found in Appendix C.

4.6. Reference Implementation with Kokkos-Kernels

To compare the proposed method from Chapter 3 with a truly parallel solver we adopted the sparse triangular routines that ship with Kokkos-Kernels 4.6. Kokkos provides a performance-portable node–level programming model; its sparse sub-package implements several SpTRSV algorithms that exploit intra-row parallelism and level scheduling on any back-end supported by Kokkos (OpenMP, CUDA, etc.) [12].

The `SPTRSVAlgorithm::SEQLVLSCHD_TP1` variant was selected because it is available for all execution spaces, performs one symbolic pass that discovers a level structure very much like the one in Section 3.4.2, and distributes rows of the same level over the OpenMP thread team while respecting data dependencies within a level by atomic updates [12]. The algorithm therefore represents the state of the art for OpenMP implementations in Kokkos-Kernels and provides an appropriate comparison for the task-graph approach of this thesis.

Just like Intel MKL, Kokkos distinguishes symbolic and numeric phases via a kernel handle:

1. Create a kernel handle: A small object kh is instantiated and configured with

   ```
   kh.create_sptrsv_handle(SPTRSVAlgorithm::SEQLVLSCHD_TP1, nrows, isLower=true);
   ```

   Here the chosen algorithm (`SEQLVLSCHD_TP1`) is a parallel level-scheduling variant, `nrows` is the matrix dimension, and the Boolean flag declares that the matrix is lower triangular.
2. Symbolic (inspector) phase: `sptrsv_symbolic(& kh, rowmap_d, entries_d, values_d);` Using the CSR structure on the device, Kokkos builds the level schedule, detects super-nodes and computes internal metadata. All results are stored inside the handle and therefore incurred only once.
3. Numeric (executor) phase: `sptrsv_solve(& kh, rowmap_d, entries_d, values_d, rhs_d, lhs_d);` The prepared schedule is applied to solve $Lx = b$ (or $Ux = b$) for the device-resident right-hand side `rhs_d`, writing the solution into `lhs_d`. This step is fully parallel and can be repeated with different b vectors while re-using the same handle.

Because the symbolic cost is amortised over all subsequent solves, the timing methodology mirrors that used for MKL and for our RACE-based solver: only the numeric phase is included in the performance figures, while the one-time inspector is measured separately.

Because Kokkos executes on the same OpenMP back-end as the task kernel, all measurements were taken with `OMP_PROC_BIND=TRUE` and `OMP_PLACES=cores` so that every benchmark sees the same binding. The wall-clock protocol mirrors MKL:

1. allocate and fill b and x once and copy them to the device views;
2. run `sptrsv_symbolic` outside the timed region;
3. repeat `sptrsv_solve` N_{rep} times;
4. record the median time t_{KK}.

Host-to-device transfers, `Kokkos::initialize` and `Kokkos::finalize` are likewise excluded from the timed section, matching the treatment of MKL's set-up overhead.

After every repetition the dense residual $r = \|b - Lx\|_2$ is assembled on the host and compared with the MKL reference. The solution returned by Kokkos is accepted when

$$\frac{|\, r_{\mathrm{KK}} - r_{\mathrm{MKL}} \,|}{r_{\mathrm{MKL}}} < 10^{-12},$$

which is the same tolerance used for the task solver (Section 4.5). All matrices given in Table 5.2 passed. The full function implementation can be found in Appendix B.

4.7. Performance Instrumentation with LIKWID

The qualitative cache–reuse arguments from Section 3.5 must be backed up by hardware-counter measurements. For this purpose the LIKWID is integrated into the build and execution workflow of the solver.

LIKWID is a light-weight suite for low–level performance monitoring on x86 processors [5]. The component `likwid-perfctr` programs the on-chip Performance-Monitoring Counters (PMCs) with a single command–line flag `-g <group>`, where a performance group is a pre-defined, architecture-specific set of events such as *L3 hits*, *L3 misses*, *executed AVX instructions*, or *DRAM bandwidth*. A second flag `-C <corelist>` selects the hardware threads to be measured.

The solver is instrumented with the `LIKWID` Marker API, which inserts a pair of ultra-low overhead system calls around the region of interest. Markers are placed inside the OpenMP parallel region so that each thread reports separate counter values and that makes it possible to specifically measure the performance of the solver implemented.

A typical run on one socket (cores 0–15) that records cache- and memory-traffic looks like

```
$ likwid-perfctr -C M0:0-15 -g L3 -m  ./tri_solve
$ likwid-perfctr -C M0:0-15 -g MEM -m ./tri_solve
```

`-m` activates the marker API, without it the whole process would be measured.

Three groups are sufficient to validate the performance model:

1. **L3**: gathers the counters `MEM_LOAD_RETIRED_L3_HIT` and `MEM_LOAD_RETIRED_L3_MISS`; the resulting hit–to–miss ratio confirms the expected high reuse of L_i during the correction step.
2. **MEM**: records the sustained DRAM bandwidth and shows if the overlapping algorithm does increase the total data volume moved.
3. **FLOPS_DP**: reports scalar and vector double-precision throughput, capturing the extra 2ℓ floating-point operations and demonstrating that the kernel is compute-bound on modern CPUs.

4.8. Build Integration and Software Dependencies

The complete solver prototype is written in modern `C++17` and is built with CMake which orchestrates the compilation.

All sources are compiled with `GCC 15.1.0`. At the time of writing this is the first GCC release whose OpenMP runtime fully supports the `affinity(...)` clause introduced in OpenMP 5.0. Earlier versions ignore the directive silently, thereby defeating the cache–local execution strategy of Section 4.4. Using 15.1.0 is therefore mandatory for the affinity–aware task kernel [10].

Dense BLAS and sparse triangular kernels used for baseline comparisons (3.5.1) are provided by Intel's oneAPI Math Kernel Library. The interface variant `lp64` is selected for 64-bit integers, and the thread layer is mapped to the OpenMP runtime already present on the system.

RACE generates the permutation that exposes block–level parallelism (3.1). It is added as an in-tree `ExternalProject` so that an unmodified upstream checkout is configured, built, and installed into the

build directory at configure time. Both RACE and LIKWID query the hardware topology through hwloc. All components rely on OpenMP for thread-level parallelism.

The resulting artefact is one relocatable binary (`tri_solve`) that pulls in RACE, MKL, hwloc, LIKWID, Kokkos-Kernels and OpenMP.

4.9. Summary

The prototype maps the proposed algorithm onto a CSR backend, parallelizes it with OpenMP tasks that mirror the theoretical dependency graph, and measures all critical kernels with LIKWID. The full source code, CMake recipes, and benchmarking scripts are archived at `https://github.com/IdsRehorst/Bachelor-Thesis-Ids-Rehorst/tree/main`. The next chapter quantifies how these design choices translate into runtime behaviour and scalability on a HPC platform.

5

Results

This chapter validates the cost model of Chapter 3 by measuring the run time, memory traffic and scaling behaviour of the proposed solver. After describing the test bed and the benchmark protocol, the observed performance is compared to Intel MKL's single–threaded `d_trsv` routine and a parallel routine based on Kokkos. In Section 5.4.4 cache reuse and the effectiveness of the affinity clause is measured and discussed.

5.1. Benchmark Platforms

All experiments were performed on two compute nodes of the DelftBlue supercomputer [17], their most relevant characteristics are summarised in Table 5.1. Each result section states explicitly which platform was used.

Table 5.1: Hardware and software environment.

	DelftBlue (Compute-p1)	DelftBlue (Compute-p2)
CPU model	Intel Xeon E5-6248R	Intel Xeon E5-6448Y
Micro-architecture	Cascade Lake	Sapphire rapids
Cores / SMT	24C / 48T	32C / 64T T
Nominal clock	3.0 GHz	2.1GHz
L3 cache	36 MiB	60 MiB
Memory	185 GiB DDR4-2933	250 GiB DDR4-2933
Compiler	gcc 15.1.0	gcc 15.1.0
MKL	oneAPI 2023.2	oneAPI 2023.2
LIKWID	5.4.1	5.4.1

5.2. Test Matrices

The test set comprises ten sparse matrices drawn from the SuiteSparse collection [4]. The full list of the matrices used for testing is given in Table 5.2. Prior to factorisation each matrix is symmetrised (if necessary), reordered by RACE, and its lower–triangular part extracted, exactly as described in Section 3.1.

The benchmark uses the lower–triangular part of A^k. Taking the cubic power is an inexpensive, purely algebraic way to inject realistic fill. For $k = 3$ these positions mimic the fill pattern created by a incomplete LU or Cholesky factorisation, so A^3 mimics the sparsity structure of a practical preconditioner application without having to run the factorisation itself.

Index	Matrix	N_r	N_{nz}	N_{nzr}	Size (MiB)
1	spinSZ12.mm	924	20356	22	0.2
2	3elt.mtx	4720	71235	15	0.8
3	crankseg_1.mtx	52804	53282167	1009	610.0
4	ship_003.mtx	121728	28378861	233	325.2
5	pwtk.mtx	217918	26474760	121	303.8
6	offshore.mtx	259789	23977337	92	275.4
7	F1.mtx	343791	149927724	436	1717.1
8	Fault_639.mtx	638802	128081733	200	1468.2
9	thermal2.mtx	1228045	13825147	11	162.9
10	Serena.mtx	1391349	319059987	229	3656.7
11	G3_circuit.mtx	1585478	16354536	11	193.2
12	nlpkkt120.mtx	3542400	50194096	14	587.9
13	delaunay_n22.mtx	4194304	45243280	10	533.8
14	channel-500x100x100-b050.mtx	464954157	183299749	96	2116.0
15	nlpkkt160.mtx	8345600	118931856	14	1392.9
16	delaunay_n23.mtx	8388608	92563506	11	1091.3
17	nlpkkt200.mtx	16240000	232232816	14	2719.6
18	delaunay_n24.mtx	16777216	180920994	10	2134.5
19	Spielman_k500_A_09.mtx	41792002	167983965	4	2081.8

Table 5.2: N_r is the number of matrix rows, and N_{nz} is the number of nonzeros after taking the lower triangular part of A^3, where A is the matrix. $N_{nzr} = N_{nr}/N_r$ is the average number of nonzeros per row. For each matrix the size is given in terms of memory when stored in the CSR format.

5.3. Methodology

Reliable performance numbers require the measurement protocol itself to be reproducible and to minimise systematic distortions. All timings in this chapter follow the same two–step procedure.

Before samples are recorded, the solver is executed once on the given matrix. This initial run brings the binary into the instruction cache, and allocates thread-private buffers inside OpenMP and MKL. The warm-up therefore eliminates one-off costs that would otherwise inflate the first measurement. After warm-up the same right-hand side is solved 100 times in immediate succession.

The elapsed time t_j is obtained with `.omp_get_wtime()`

Repeating the kernel masks short-lived perturbations. To prevent noise from other processes, the benchmarks were run on an exclusive node.

5.4. Results

5.4.1. Parallel Strong Scaling

Parallel efficiency was assessed on a single DelftBlue compute-p1 node for thread counts $p \in \{1, 2, \ldots, 24\}$. For each solver the strong–scaling factor is reported as

$$S(p) = \frac{t_1}{t_p}, \qquad t_p = \text{median run time with } p \text{ OpenMP threads.}$$

Figures 5.1 and 5.2 show the curves for the five matrices with the largest non-zero counts from Table 5.2. Restricting the plot to these cases avoids an unreadable forest of lines while still covering the most demanding workloads, the smaller matrices exhibit the same qualitative trend but saturate earlier due to their limited concurrency.

For the task–graph solver (Fig. 5.1) the speed-up ranges from 1.28 (`Serena`) to 1.45 (`Spielman_k500_A_09.mtx`). Across all cases the curve flattens once $p \gtrsim 6$ because *Phase 2* (see Section 3.4.2) becomes the critical path and thus further threads can only steal leftover provisional tasks whose contribution to total run time is already marginal.

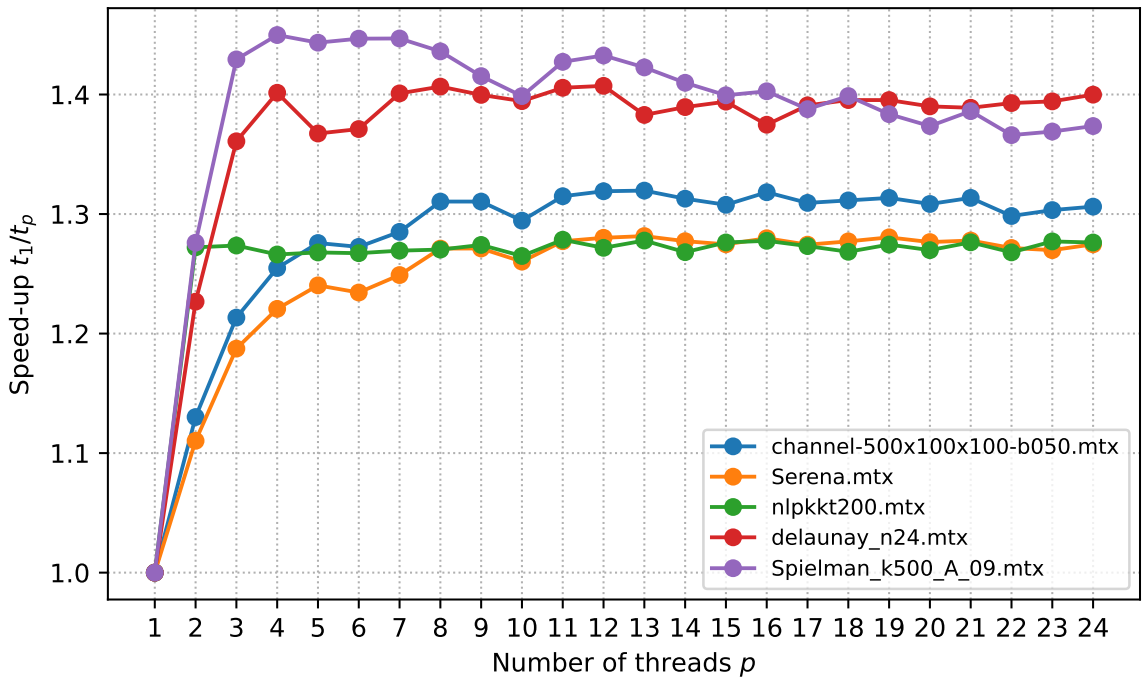


Figure 5.1: Strong–scaling of the task-based bi-block solver (five largest matrices, see Table 5.2), executed on the compute-p1 node of DelftBlue.

Kokkos-Kernels exhibits a larger head-room (Fig. 5.2): $S(24)$ varies between 1.75 and 2.8 and saturation is only reached around $p = 16$.

A notable outlier is `channel-500x100x100-b050.mtx`, whose speed-up curve alternates between two distinct plateaus, an indication that Kokkos-Kernels may toggle between two internal solve strategies for this matrix.

Intel MKL is not included in the scaling plots because `mkl_sparse_trsv` is a sequential kernel and therefore provides no meaningful parallel speed-up curve. Its single-thread timing will be used as a performance baseline in the next sections.

5.4.2. Run-Time Versus Problem Size

Figure 5.3 juxtaposes the solve times of the three solvers (Intel MKL, the task–graph implementation, and the Kokkos–Kernels reference) against the matrix size (measured by the number of non-zeros N_{nz}). Each panel fixes the thread count $p \in \{1, 6, 12, 24\}$; the horizontal axis is shared so that slopes can be compared directly. All measurements were obtained on the compute-p1 node of DelftBlue (Table 5.1).

At $p = 1$ (upper-left panel) MKL unsurprisingly delivers the fastest solves; its algorithmic overhead is modest and the code is heavily optimised for serial performance. The task solver follows the same trend but incurs a higher constant cost because symbolic permutation and task management are executed on the critical path. Kokkos–Kernels is the slowest method for most matrices, showing a larger overhead.

When additional threads become available ($p = 6$, 12, 24) the curves of the parallel solvers turn downwards, while the single–threaded MKL line stays in place. For the task solver the improvement is noticeable up to about six threads and then flattens, in line with the strong–scaling results discussed later in Section 5.4.1. Kokkos–Kernels continues to shorten run-time up to $p \approx 16$ and then levels off where the turning point depends on the matrix structure.

A final observation from Figure 5.3 is that for small matrices ($N_{nz} \lesssim 10^7$) the one-threaded Intel MKL routine is consistently the fastest of the three solvers even when parallel resources are available. The higher constant overheads of the task graph (task creation) and of Kokkos-Kernels (kernel-handle

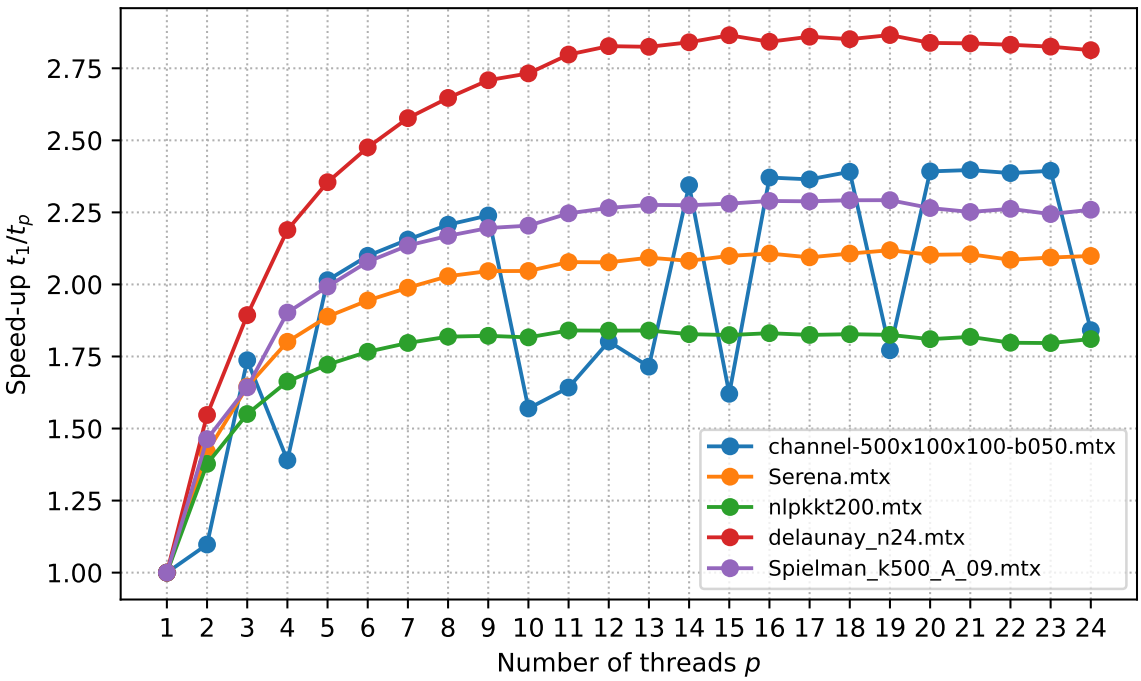


Figure 5.2: Strong–scaling of the Kokkos-Kernels `sptrsv` implementation for the same matrices and hardware as in Fig. 5.1.

initialisation, generic data structures) outweigh their potential to exploit additional threads. Only when the matrix exceeds roughly ten million non-zeros does the parallel execution of the task solver or Kokkos become beneficial relative to MKL's highly tuned serial path.

Because absolute run-times obscure the relative gains, and because the performance of the solvers lie very close together, the next section converts these measurements into speed-up curves, which better expose algorithmic scalability.

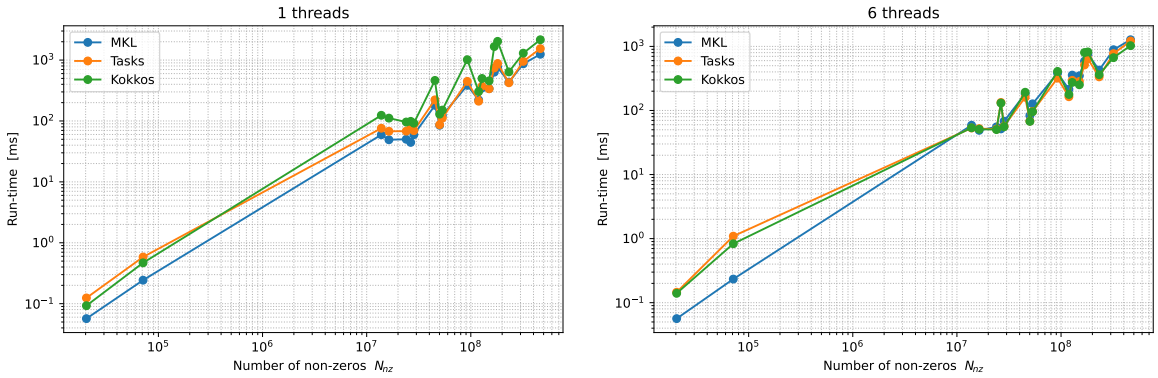
5.4.3. Cross–Solver Comparison

Figure 5.4 shows the ratio $t_{\text{MKL}}/t_{\text{tasks}}$ while Figure 5.5 reports $t_{\text{Kokkos}}/t_{\text{tasks}}$. Each curve corresponds to a fixed thread count $p \in \{1, 6, 12, 24\}$ and is plotted over the non–zero count of the test matrices. Measurements were once again performed on an exclusive compute-p1 node on DelftBlue (Table 5.1).

Because Intel MKL's triangular kernel is single-threaded, the ratio $t_{\text{MKL}}/t_{\text{tasks}}$ grows with p making our implementation up to an order of magnitude faster at $p = 24$. For $p = 1$ the Intel MKL solver is in almost all cases the faster method as should be expected.

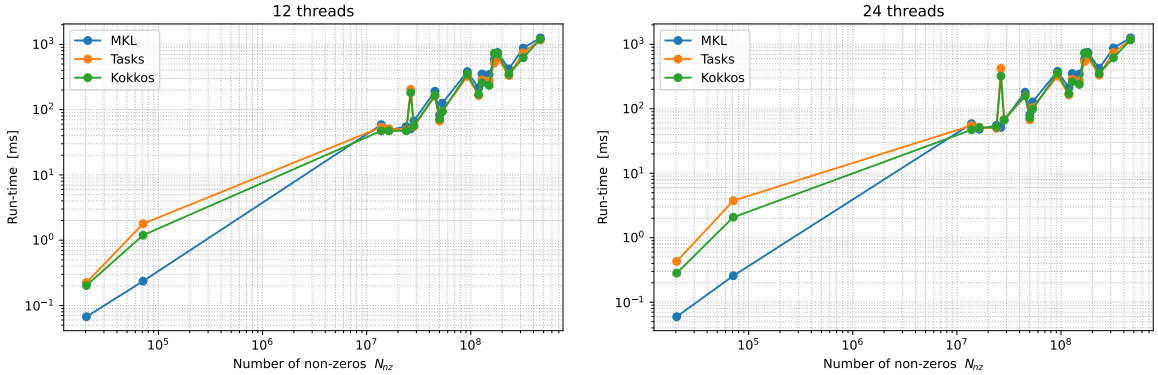
In Figure 5.4 there is one very notable outlier to the general performance trend, this turns out to be caused by matrix `Ship_003`. `Ship_003` is the only test matrix that is symmetric positive-definite in its original form and therefore a direct Cholesky candidate. Its factor L develops large, nearly dense super-nodes and RACE collapses these into just a few bulky diagonal blocks, leaving virtually no task-level parallelism. Our two–pass task kernel still incurs its bookkeeping overhead, but with little memory latency to overlap, this cost dominates the run time. Conversely, `mkl_sparse_d_trsv` detects the dense structure and switches to a vectorised dense micro-kernel that streams efficiently through contiguous data, even in a single thread. Hence `Ship_003` falls far below the general speed-up trend: MKL outperforms the task solver and additional threads cannot close the gap. The anomaly delineates the solver's scope: it excels on irregular, latency-dominated triangular factors, but its edge disappears when the factor resembles a dense panel already handled efficiently by a tuned serial routine.

The curves in Figure 5.5 lie much closer to the horizontal axis, staying between 0.75 and 1.5 for almost the entire size range when more then one thread is used. This confirms that after the strong-scaling "plateau" of Figure 5.1 is reached both parallel algorithms become memory-bound and therefore deliver



(a) $p = 1$ thread

(b) $p = 6$ threads



(c) $p = 12$ threads

(d) $p = 24$ threads

Figure 5.3: Run-time versus problem size for the three solvers. Each line connects matrices in ascending N_{nz} order and both axes use logarithmic scales. Measurements performed on the compute-p1 node of DelftBlue.

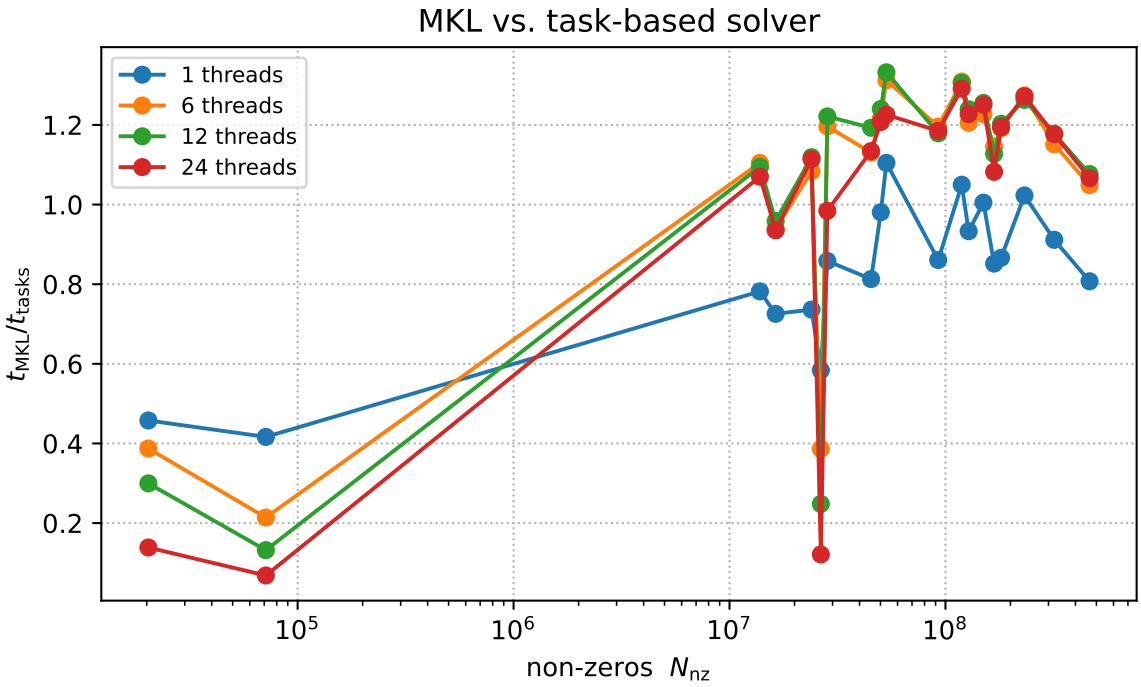


Figure 5.4: Relative speed-up $\frac{t_{MKL}}{t_{tasks}}$. Points above 1 indicate that the task-based solver outperforms Intel MKL and points below 1 indicate the opposite. Results are shown for $p = 1$, 6, 12, and 24 threads, with matrices ordered by increasing nnz.

very similar absolute performance. The apparent advantage of Kokkos at $p \geq 12$ must therefore be read with caution: it reflects the fact that our method saturates a few threads earlier and subsequently shows little change, whereas the Kokkos implementation continues to shorten its run time until about $p \approx 16{-}18$. When the arithmetic workload is tiny (leftmost data points) the Kokkos solver is clearly faster for $p \geq 1$.

The two ratios underline that speed-up is a relative metric: although Kokkos attains larger strong-scaling factors (Figure 5.2), its advantage in wall-clock time shrinks once both solvers are limited by memory bandwidth. Put differently, the task-graph formulation reaches the memory-bound regime earlier, so its speed-up curve becomes flat before Kokkos has exhausted all available cores. For large matrices and $p \geq 12$ the two implementations therefore exhibit virtually identical run times.

Across the large-matrix test set no single solver dominates universally: for some inputs the task-based bi-block method is faster, for others the Kokkos implementation wins. The relative ranking therefore depends on the individual sparsity pattern rather than on matrix size alone.

5.4.4. LIKWID Results

All LIKWID measurements were taken on the DelftBlue compute-p2 node (5.1). The region of our solver was instrumented with the *L3*, *L2/L2CACHE* and *DATA* event groups. The *MEM* group is unavailable on this platform because current LIKWID versions cannot program the Intel Sapphire Rapids integrated-memory-controller, so the counters `CAS_COUNT_RD/WR` return 0. Consequently, the analysis below focuses on the on-chip hierarchy only.

LIKWID was used to measure the difference in performance of the task-based model implemented with OpenMP in Section 4.3, before the affinity clause was added to the OpenMP directives. To interpret the cache behaviour we restrict the discussion to a single matrix that cannot reside in any on-chip cache and therefore stresses the reuse mechanisms most: `nlpkkt200` (see Table 5.2). After the A^3 expansion and triangular extraction the factor is sure to not fit in the cache memory of *DelftBlue*.

With the LIKWID `L3`, `L2CACHE` and `DATA` groups the triangular solve of our task kernel (16 threads)

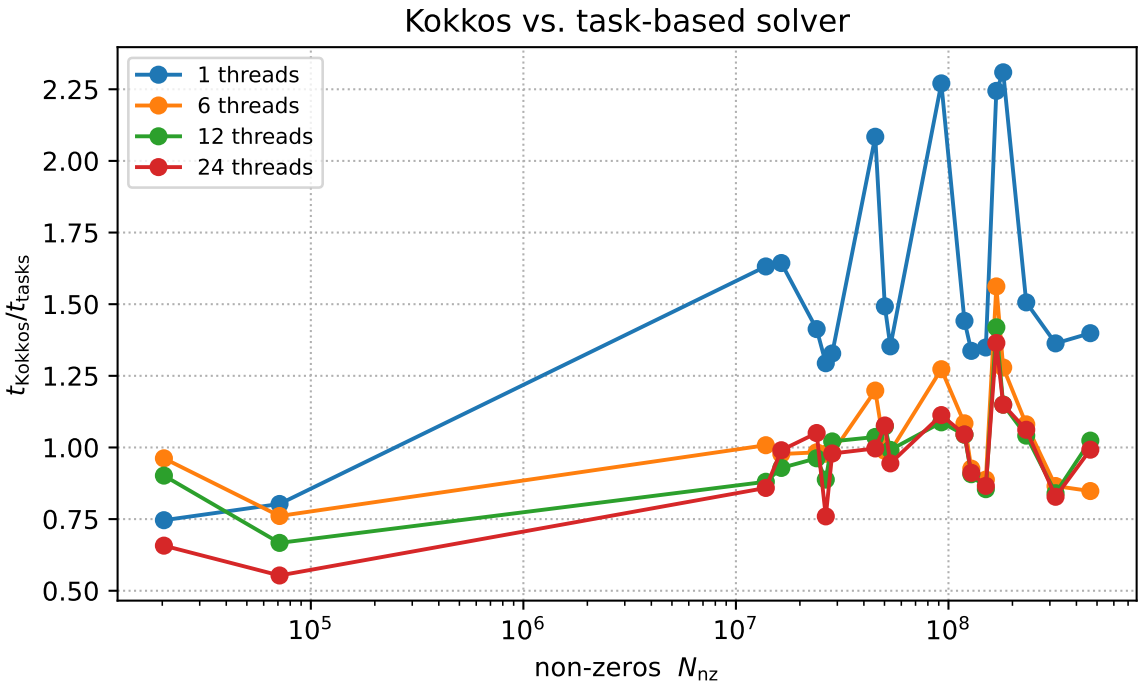


Figure 5.5: Relative speed-up $\frac{t_{\text{Kokkos}}}{t_{\text{tasks}}}$. Values above 1 mean the task-based solver is faster than the Kokkos-Kernels implementation, while values below 1 favour Kokkos.

reports

$$\text{L3 loads} = 7.1\ \text{GB}, \quad \text{L3 miss ratio} = 12\%, \quad \text{L2 miss ratio} = 91\%.$$

The missing `MEM` group on Sapphire Rapids prevents direct DRAM-traffic measurements; nonetheless the figures are consistent with an in-LLC working set. Almost every line requested by a core is absent from its private L2, but nine out of ten are satisfied in the shared LLC, so only ≈ 0.85 GB of the 7.1 GB finally spill to memory.

A ~90 % L2 miss rate means that most provisional and correction tasks run on different cores: the provisional pass of block i brings L_i and x_i into its private cache, the correction pass is then stolen by another worker, and the data are fetched again from LLC. This behaviour confirms the suspicion that the default work-stealing policy weakens the intended producer–consumer affinity. When the same thread executed both phases in quick succession one would expect up to 50 % L2 hits, because an entire L_i ($\leq$2 MiB) fits into a single private cache.

In Section 4.4 the `affinity` clause was added to the OpenMP task directives to keep each correction task on the same core that had just produced its matching provisional result, thereby preserving the block data in the private L2 cache. Owing to time constraints in the final week of the project, a second LIKWID counter study could not be performed; only wall-clock timings were recorded.

Those timings are nevertheless decisive. For every matrix and for all thread counts $p \in \{1, 2, 3, 6, 9, 12\}$ the version with the `affinity` hint outperforms the earlier non-affine implementation. Figure 5.6 plots the ratio

$$\text{speed-up}_{\text{aff}} = \frac{t_{\text{no-affinity}}}{t_{\text{affinity}}},$$

so values above 1 indicate a benefit from affinity scheduling. The average improvement ranges from $\sim 1.1\times$ for single-thread runs—where no scheduling decisions are necessary—up to almost $\sim 2\times$ for nine threads. This confirms the intuition from Section 4.4: suppressing work-stealing along the tightly

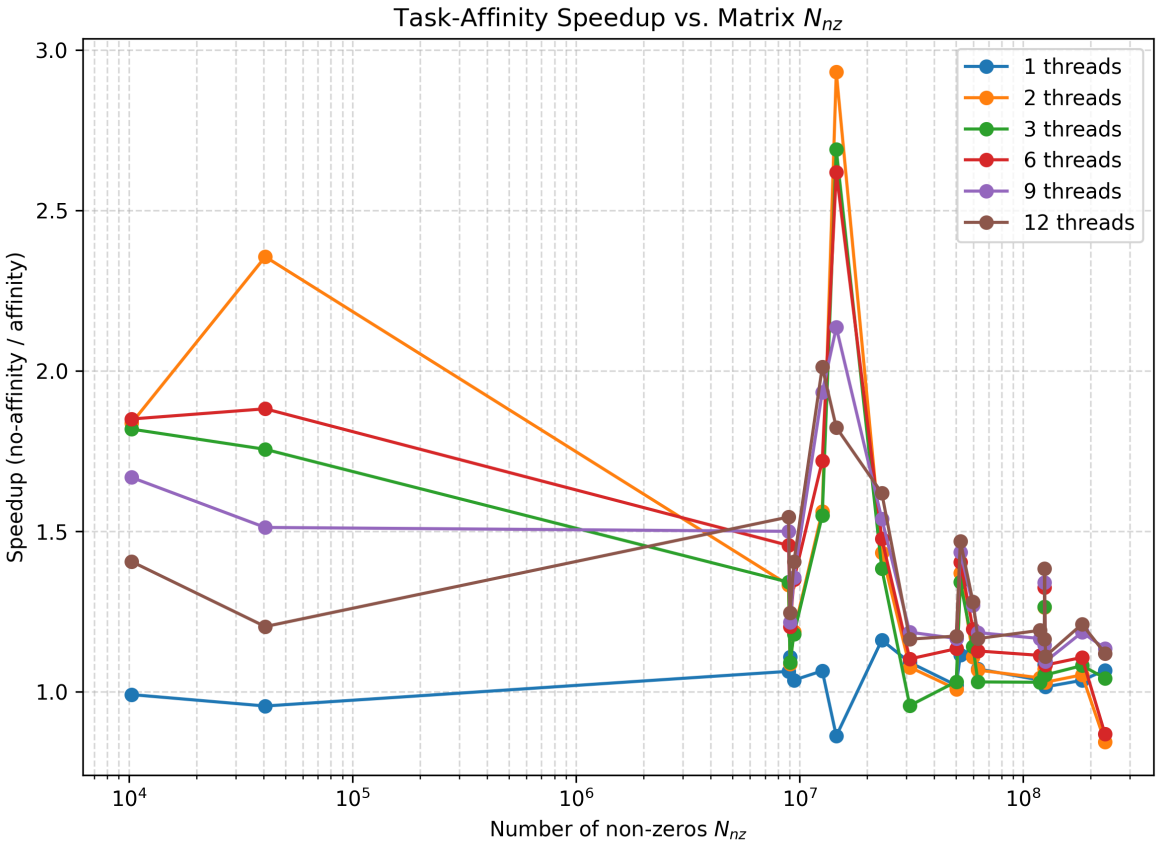


Figure 5.6: Relative speed-up between the task based solver when implemented with and without affinity clause. Speed-up is given by solve time of implementation without affinity divided by the implementation with affinity for p $\in \{1, 2, 3, 6, 9, 12\}$ threads.

coupled correction chain leads to markedly better cache-line reuse and shorter overall solve time, even though the `affinity` clause is merely a hint and the OpenMP runtime may, in principle, choose a different mapping.

5.5. Reproducibility and Data Availability

All raw CSV measurements, plotting scripts, and the exact solver sources that produced the figures of this chapter are publicly available at `https://github.com/IdsRehorst/Bachelor-Thesis-Ids-Rehorst`. The repository's `tests/` directory contains `benchmark_`$< p >$`.csv` files – one per thread count – that hold the median run time of every matrix and solver variant, ready for independent analysis or alternative visualisations.

6

Conclusion and Recommendations

6.1. Conclusions

The thesis set out to examine whether redundant computation can accelerate sparse triangular solves by (i) exposing parallelism, (ii) increasing cache reuse and (iii) delivering tangible benefits over established vendor libraries. The objectives are now revisited after the results found in Chapter 5.

By recasting the forward substitution into a two-phase block–bidiagonal schedule and mapping that schedule to OpenMP tasks, the solver sustained concurrent progress on up to six CPU cores before the critical correction chain became the only bottleneck. On the five largest matrices the strong-scaling factor reached $1.3 - 1.45$ already at $p = 6$ threads (Fig. 5.1). Although this absolute speed-up is modest, it is achieved with considerably fewer threads than Kokkos-Kernels, which continues to scale until about $p = 16$ (Fig. 5.2). The experiment therefore confirms that the redundant formulation does indeed reveal exploitable parallelism in SpTRSV.

Hardware counters taken prior to introducing the `affinity` clause showed an L_2 miss rate above 90 %, indicating that OpenMP's default work-stealing repeatedly moved paired tasks onto different cores. Adding an affinity hint reduced the median run time by up to a factor of two (Fig. 5.6), proving that the producer–consumer locality envisioned in the cost model can be realised in practice. Although a second LIKWID study was not possible within the allotted time, the consistent wall-clock improvement across all matrices substantiates the claim that cache reuse was substantially improved.

Against Intel MKL the task solver is constrained by MKL's exceptionally fast single-thread path; nevertheless at $p = 24$ it is up to an order of magnitude faster because MKL is sequential by design (Fig. 5.4). When compared to the parallel reference in Kokkos-Kernels the picture is more balanced. Our implementation reaches its memory-bandwidth limit after six to eight threads, whereas Kokkos continues to shorten run time until the core count approaches sixteen. The resulting speed-ups therefore hover around unity for large matrices (Fig. 5.5), i.e. both solvers are equally fast once they are bandwidth bound. The only systematic deviation occurs for `Ship_003`, whose dense Cholesky-like structure favours MKL's dense micro-kernel and offers little block parallelism, underscoring the limits of the approach for well structured SPD systems.

The study demonstrates that a redundant two-pass algorithm, even in a first implementation, can rival highly optimised library code while requiring only a handful of threads and no architecture-specific tuning. Its advantage is most pronounced for large, irregular matrices where memory latency dominates, and it diminishes for small or Cholesky-friendly factors that fit into cache. Given the simplicity of the kernel and the conservative thread counts used, the results suggest that further optimisation could extend the performance range still further.

6.2. Recommendations

While the present study establishes that redundant two-phase scheduling can match the performance of vendor libraries on contemporary multi-core CPUs, it also exposes several directions in which the

prototype could be extended and strengthened.

Inside each diagonal block the prototype still calls a plain row-by-row SpTRSV (`serialSpTRSV`). Substituting this routine by a high-performance micro-kernel, e.g. a blocked ICC(0) solve or a small-BLR triangular kernel, would reduce solve time and improve numerical stability when pivots are ill-conditioned. A drop-tolerant or mixed-precision variant could trade accuracy for bandwidth on bandwidth-bound blocks.

Although the `affinity` hints introduced in Section 4.4 reduce work stealing, the OpenMP standard still provides no formal guarantee that a producer–consumer task pair will execute on the same core. Consequently, cache residency remains a heuristic decision of the runtime and can be lost under load imbalance or NUMA pressure. Future work should therefore explore task–scheduling strategies, whether via alternative runtimes or custom policies, that treat cache reuse as a hard constraint rather than a best-effort optimisation.

So far RACE is used in its default distance-1 mode, producing a bi-block diagonal pattern. Allowing distance-2 or distance-k colourings would generate wider pipelines and expose additional parallelism once Phase 2 becomes the critical path.

A more robust code base should shield users from matrices that violate assumptions such as explicit diagonals, symmetric sparsity or non-singular blocks. Integrating run-time guards, automated unit tests and continuous integration would harden the solver for production use and ensure that performance regressions are caught early.

Exploring these avenues would deepen the understanding of redundant triangular solves, extend their applicability to truly massive problems, and sharpen the performance edge over established vendor libraries.

References

[1] Christie Alappat et al. "A Recursive Algebraic Coloring Technique for Hardware-efficient Symmetric Sparse Matrix-vector Multiplication". In: *ACM Trans. Parallel Comput.* 7.3 (June 2020). ISSN: 2329-4949. DOI: 10.1145/3399732. URL: https://doi.org/10.1145/3399732.

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Tech. rep. OpenMP Architecture Review Board, 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[3] L. M. Carvalho. "A performance comparison of linear algebra libraries for sparse matrix-vector product". In: *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics* (2015). DOI: 10.5540/03.2015.003.01.0116. URL: https://doi.org/10.5540/03.2015.003.01.0116.

[4] Timothy A. Davis and Yifan Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663.

[5] Thomas Gruber et al. *LIKWID*. Version 5.4.1. 2024. DOI: 10.5281/zenodo.14364500. URL: https://doi.org/10.5281/zenodo.14364500.

[6] *Intel® Math Kernel Library for Linux* Developer Guide*. Revision 068 Intel® MKL 2020. Intel Corporation. Santa Clara, CA, 2020. URL: https://cdrdv2-public.intel.com/671193/mkl-2020-developer-guide-linux.pdf.

[7] David Levinthal. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. Tech. rep. Intel Corporation, 2010. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf.

[8] Sirine Marrakchi and Heni Kaaniche. "Solving Sparse Triangular Linear Systems: A Review of Parallel and Distributed Solutions". In: *Intelligent Systems Design and Applications*. Ed. by Ajith Abraham et al. Cham: Springer Nature Switzerland, 2024, pp. 440–449. ISBN: 978-3-031-64850-2.

[9] OpenMP Architecture Review Board. *OpenMP Application Programming Interface — Version 5.0*. Specification Version 5.0. OpenMP Architecture Review Board. Nov. 2018. URL: https://www.openmp.org/specifications/.

[10] OpenMP Architecture Review Board. *OpenMP® Compilers & Tools*. https://www.openmp.org/resources/openmp-compilers-tools/. Accessed 27 June 2025. 2025.

[11] Jongsoo Park et al. "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver". In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Cham: Springer International Publishing, 2014, pp. 124–140. ISBN: 978-3-319-07518-1.

[12] Sivasankaran Rajamanickam et al. *Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels*. 2021. arXiv: 2103.11991 [cs.MS]. URL: https://arxiv.org/abs/2103.11991.

[13] Thomas Rauber and Gudula Rünger. *Parallel Programming*. Cham: Springer International Publishing, 2023. DOI: 10.1007/978-3-031-28924-8.

[14] Arch Robison. *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*. WG21 Technical Report N3872. Doc. No. N3872. ISO/IEC JTC1/SC22/WG21, Jan. 2014. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf.

[15] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[16] Barry Smith and Hong Zhang. "Sparse triangular solves for ILU revisited: Data layout crucial to better performance". In: *IJHPCA* 25 (Nov. 2011), pp. 386–391. DOI: 10.1177/1094342010389857.

[17] TU Delft High-Performance Computing Centre (DHPC). *Description of the DelftBlue system*. Last updated 2024-09-01, retrieved. Sept. 2024. URL: https://doc.dhpc.tudelft.nl/delftblue/DHPC-hardware/#description-of-the-delftblue-system (visited on 06/12/2024).

[18] Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan Ellingwood. "Performance Portable Supernode-based Sparse Triangular Solver for Manycore Architectures". In: *Proceedings of the 49th International Conference on Parallel Processing*. ICPP '20. Edmonton, AB, Canada: Association for Computing Machinery, 2020. ISBN: 9781450388160. DOI: 10.1145/3404397.3404428. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3404397.3404428.

A

Block Bidiagonal Task Based Solver

The full implementation of the block bidiagonal task based solver used to solve sparse triangular system is given below.

Listing A.1: Full implementation of the `blockBiDiagSolveTasks` kernel.

```
void solver::blockBiDiagSolveTasks(const sparsemat&           B,
                                   const std::vector<int>&    stagePtr,
                                   const std::vector<double>& b,
                                   std::vector<double>&       x)
{
    const int k = int(stagePtr.size()) - 1;
    const int N = B.n;
    x.assign(N, 0.0);

    /* raw pointers so that OpenMP array-section syntax works */
    double       *xp = x.data();
    const double *bp = b.data();

#pragma omp parallel default(none) shared(B,stagePtr,bp,xp,k)
{
#pragma omp single
{
    /* ---------------- Phase 1 : provisional solves ---------------- */
    for (int i = 0; i < k; ++i) {
        const int r0 = stagePtr[i];
        const int r1 = stagePtr[i+1];
        const int m  = r1 - r0;                /* block size            */

        /* length in the array-section must be  r1-r0,  not the last index */
#pragma omp task depend(out: xp[r0 : m])
affinity(r0)
firstprivate(r0,r1,m)
        {
            std::vector<double> rhs(m), xi(m);

            /* RHS = b_i */
            for (int j = 0; j < m; ++j)
                rhs[j] = bp[r0 + j];

            /* solve L_i * x = rhs */
            for (int ii = 0; ii < m; ++ii) {
                int    row  = r0 + ii;
                double sum  = rhs[ii];
                double diag = 1.0;

                for (int p = B.rowPtr[row]; p < B.rowPtr[row+1]; ++p) {
                    int c = B.col[p];
                    if      (c <  r0) continue;            /* belongs to B_i   */
                    else if (c <  row) sum  -= B.val[p] * xi[c - r0];
```

```
                    else if (c == row) diag  = B.val[p];
                }
                assert(std::abs(diag) > 1e-30);
                xi[ii] = sum / diag;
            }

            /* write provisional result */
            for (int j = 0; j < m; ++j) xp[r0 + j] = xi[j];
        }
    }

    /* ---------------- Phase 2 : correction solves ----------------- */
    for (int i = 1; i < k; ++i) {
        const int r0 = stagePtr[i];
        const int r1 = stagePtr[i+1];
        const int m  = r1 - r0;

#pragma omp task  depend(in:    xp[stagePtr[i-1] : stagePtr[i]-stagePtr[i-1]]) \
                  depend(inout: xp[r0            : m])                         \
                  affinity(r0)
                  firstprivate(r0,r1,m)
        {
            std::vector<double> rhs(m), xi(m);

            /* RHS = b_i - B_i * x_{i-1} */
            for (int ii = 0; ii < m; ++ii) {
                int    row = r0 + ii;
                double sum = bp[row];

                for (int p = B.rowPtr[row]; p < B.rowPtr[row+1]; ++p) {
                    int c = B.col[p];
                    if (c < r0) sum -= B.val[p] * xp[c];
                }
                rhs[ii] = sum;
            }

            /* solve L_i*x = rhs */
            for (int ii = 0; ii < m; ++ii) {
                int    row  = r0 + ii;
                double sum  = rhs[ii];
                double diag = 1.0;

                for (int p = B.rowPtr[row]; p < B.rowPtr[row+1]; ++p) {
                    int c = B.col[p];
                    if      (c <  r0) continue;            /* already in RHS   */
                    else if (c <  row) sum  -= B.val[p] * xi[c - r0];
                    else if (c == row) diag  = B.val[p];
                }
                assert(std::abs(diag) > 1e-30);
                xi[ii] = sum / diag;
            }

            /* write corrected result */
            for (int j = 0; j < m; ++j) xp[r0 + j] = xi[j];
        }
    }
    /* implicit taskwait here */
} /* single */
} /* parallel */
}
```

B

Parallel SpTRSV implementation with Kokkos

The full implementation of the Kokkos based solver used as a reference to the method presented in Appendix A is given below.

Listing B.1: Full implementation of the `kokkosSpTRSV` kernel.

```
    #include <Kokkos_Core.hpp>
#include <KokkosSparse_sptrsv.hpp>
#include <KokkosKernels_Handle.hpp>
#include <chrono>          // <-- for timing

double solver::kokkosSpTRSV(const sparsemat&           B,
                            const std::vector<double>& b,
                            std::vector<double>&       x)
{
  using exec_space = Kokkos::DefaultExecutionSpace;          // OpenMP here
  using mem_space  = typename exec_space::memory_space;

  using size_type  = int;      // match your CSR index types
  using lno_type   = int;
  using scalar     = double;

  const size_type n   = B.n;
  const size_type nnz = static_cast<size_type>(B.val.size());

  /* ------------ host views (build once per call) ---------------- */
  using RowH = Kokkos::View<size_type*, Kokkos::HostSpace>;
  using ColH = Kokkos::View<lno_type*,  Kokkos::HostSpace>;
  using ValH = Kokkos::View<scalar*,    Kokkos::HostSpace>;

  RowH row_h("row_h", n + 1);
  ColH col_h("col_h", nnz);
  ValH val_h("val_h", nnz);

  for (size_type i = 0; i <= n;  ++i) row_h(i) = B.rowPtr[i];
  for (size_type p = 0; p <  nnz; ++p) {
    col_h(p) = B.col[p];
    val_h(p) = B.val[p];
  }

  /* ------------ device mirrors ---------------------------------- */
  using RowD = Kokkos::View<size_type*, mem_space>;
  using ColD = Kokkos::View<lno_type*,  mem_space>;
  using ValD = Kokkos::View<scalar*,    mem_space>;
  using VecD = Kokkos::View<scalar*,    mem_space>;

  RowD row_d("row_d", n + 1);   ColD col_d("col_d", nnz);
```

```
  ValD val_d("val_d", nnz);     VecD b_d  ("b_d",   n);
  VecD x_d  ("x_d",   n);

  Kokkos::deep_copy(row_d, row_h);
  Kokkos::deep_copy(col_d, col_h);
  Kokkos::deep_copy(val_d, val_h);

  { // copy RHS once
    auto b_h = Kokkos::create_mirror_view(b_d);
    for (size_type i = 0; i < n; ++i) b_h(i) = b[i];
    Kokkos::deep_copy(b_d, b_h);
  }

  /* ------------ kernel-handle + symbolic ------------------------ */
  using KH = KokkosKernels::Experimental::KokkosKernelsHandle<
              size_type, lno_type, scalar,
              exec_space, mem_space, mem_space>;

  KH kh;
  kh.create_sptrsv_handle(
        KokkosSparse::Experimental::SPTRSVAlgorithm::SEQLVLSCHD_TP1,
        n, /*is_lower =*/ true);

  KokkosSparse::Experimental::sptrsv_symbolic(&kh, row_d, col_d, val_d);
  exec_space().fence();   // done with symbolic part

  /* ------------ NUMERIC solve (timed) --------------------------- */
  auto t0 = std::chrono::high_resolution_clock::now();
  KokkosSparse::Experimental::sptrsv_solve(&kh,
          row_d, col_d, val_d,
          b_d,  x_d);

  exec_space().fence();
  auto t1 = std::chrono::high_resolution_clock::now();
  const double t_ms =
      std::chrono::duration<double, std::milli>(t1 - t0).count();

  /* ------------ copy result back -------------------------------- */
  auto x_h = Kokkos::create_mirror_view(x_d);
  Kokkos::deep_copy(x_h, x_d);
  x.assign(x_h.data(), x_h.data() + n);

  kh.destroy_sptrsv_handle();
  return t_ms;                    // <----  numeric time only
}
```

C

SpTRSV Implementation with Intel MKL

The full implementation of the Intel MKL based solver used as a reference to the method presented in Appendix A is given below.

Listing C.1: Full implementation of the `MKL SpTRSV` kernel.

```
    void solver::mklTriSolve(const sparsemat &B, bool lower,
                         const std::vector<double> &b,
                         std::vector<double> &x)
{
    int n = B.n;

    // --- check if every row has an explicit diagonal --------------------
    bool explicitDiag = true;
    for (int r = 0; r < n && explicitDiag; ++r) {
        bool found = false;
        for (int p = B.rowPtr[r]; p < B.rowPtr[r + 1]; ++p)
            if (B.col[p] == r) { found = true; break; }
        explicitDiag = found;
    }


    sparse_matrix_t A = nullptr;
    matrix_descr desc{};
    desc.type = SPARSE_MATRIX_TYPE_TRIANGULAR;
    desc.mode = lower ? SPARSE_FILL_MODE_LOWER : SPARSE_FILL_MODE_UPPER;
    desc.diag = explicitDiag ? SPARSE_DIAG_NON_UNIT : SPARSE_DIAG_UNIT;

    std::vector<MKL_INT> ia(n + 1);  std::vector<MKL_INT> ja(B.col.size());
    for (int i = 0; i <= n; ++i) ia[i] = B.rowPtr[i];
    for (size_t k = 0; k < B.col.size(); ++k) ja[k] = B.col[k];

    // Init Likwid marker for measuring perfomance
    LIKWID_MARKER_START("MKL");

    mkl_sparse_d_create_csr(&A, SPARSE_INDEX_BASE_ZERO,
                            n, n, ia.data(), ia.data() + 1,
                            ja.data(), const_cast<double*>(B.val.data()));
    mkl_sparse_optimize(A);

    x.assign(n, 0.0);
    mkl_sparse_d_trsv(SPARSE_OPERATION_NON_TRANSPOSE,
                      1.0, A, desc,
                      b.data(), x.data());
    mkl_sparse_destroy(A);
    LIKWID_MARKER_STOP("MKL");
}
```