POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Performance Optimization of a Legacy FEM Solver

Tesi di Laurea Magistrale in
HPC Engineering

Author: **Alessandro Colombo**

Student ID: 976041
Advisor: Prof. Luca Formaggia
Co-advisors: Prof. Alessandro Croce
Academic Year: 2025-2026

Contents

1 | Introduction

1.1. Evolution of High Performance Computing and Legacy Scientific Software

Over the last two decades, High Performance Computing (HPC) has undergone a profound transformation. Computational systems that were once based almost exclusively on sequential or moderately parallel CPU architectures have progressively evolved toward highly heterogeneous platforms, in which multicore processors, vectorized instructions, GPUs, and specialized accelerators coexist to provide unprecedented computational power. At the same time, numerical simulation has become increasingly central to engineering and scientific research, requiring software that efficiently exploits modern hardware while maintaining reliability, scalability, and numerical accuracy.

Many engineering simulation codes still in use today were developed between the early 2000s and the beginning of the 2010s. Their design was heavily influenced by the hardware and programming paradigms available at the time, when applications were mainly optimized for traditional x86 CPU architectures, with limited focus on large-scale parallelism, memory locality, or heterogeneous computing. Technologies such as GPUs and advanced accelerator-based systems were either unavailable or not yet widely adopted in scientific computing.

Despite their age, these legacy codes still retain significant scientific and engineering value, since they incorporate years of validated numerical models, specialized algorithms, and domain-specific knowledge. For this reason, the modernization of legacy scientific software has become an important challenge in modern High Performance Computing research.

This study aims to contribute to the ongoing discussion regarding software sustainability in scientific computing. As HPC architectures continue to evolve rapidly, the gap between legacy software and modern hardware tends to increase. Many scientific communities face the same dilemma: whether to preserve and evolve existing computational tools or abandon them in favor of entirely new solutions. Understanding the technical, computational, and organizational implications of software modernization is therefore essential not only for this specific project but also for the future management of scientific software infrastructures.

1.2. The idea behind this thesis

The main objective of this thesis is to investigate whether a legacy numerical simulation code can be effectively modernized and adapted to contemporary hardware architectures in order to achieve significant improvements in computational performance and resource efficiency. The project explores the possibility of evolving and optimizing an established codebase by analyzing its structure, identifying its computational bottlenecks, and introducing targeted improvements that exploit modern HPC technologies.

This approach raises several important questions. First, it is necessary to understand whether software originally conceived for older architectures can realistically benefit from modern parallel computing paradigms. The introduction of multicore processors, advanced cache hierarchies, SIMD vectorization, GPUs, and accelerator-based computing has radically changed the way high-performance applications are designed and optimized. Algorithms and data structures that were considered efficient fifteen or twenty years ago may now represent severe limitations due to inefficient memory access patterns, poor scalability, or insufficient exploitation of hardware parallelism.

A second aspect concerns the practical feasibility of modernization. Updating a legacy code is not simply a matter of recompiling it on newer hardware. In many cases, significant refactoring may be required to improve data locality, reduce memory overhead, expose parallelism, and redesign computational kernels. One of the key goals of this work is to evaluate the effort required to modernize an existing scientific application and whether the resulting performance gains justify the investment.

Another fundamental point is the comparison between modernization and complete redevelopment. Rewriting a scientific code from scratch may offer the opportunity to adopt modern programming paradigms and optimized software architectures from the start. However, this approach also introduces long development cycles and validation difficulties. Conversely, incremental modernization may preserve the reliability and scientific consistency of the original code while progressively improving its efficiency and adaptability. Determining which of these two strategies is more advantageous represents one of the central motivations of the project.

The project seeks to understand how legacy engineering software can adapt to the modern HPC ecosystem, what limitations emerge during the optimization process, and what trade-offs characterize the transition from traditional CPU-oriented implementations to contemporary parallel computing environments.

1.3. Case Study: Cp-Lambda Aeroelastic Code

The case study considered in this work is Cp-Lambda, an aeroelastic finite element multibody code developed at the Politecnico di Milano. The software is written primarily in C and supported by a set of MATLAB scripts that handle preprocessing and postprocessing. Using these tools, the code is able to ingest detailed structural and aerodynamic input data describing complex mechanical systems and return a set of intermediate parameters that are subsequently used by the numerical solver to evaluate stresses, strains, and

overall structural response across the entire configuration.

Cp-Lambda is designed as a flexible multibody framework capable of representing different structural configurations and actuation mechanisms. Its formulation enables the analysis of coupled aeroelastic phenomena, making it suitable for the study of flexible structures subjected to aerodynamic loads. In this work, the focus is placed on a wind turbine configuration, specifically the DLC 1.1 (Design Load Case 1.1), which represents a fundamental operating condition used in aeroelastic and structural assessment of wind energy systems.

From a computational perspective, Cp-Lambda relies on a nonlinear solution strategy based on the Newton–Raphson method to iteratively solve the governing equilibrium [6]. This approach, while robust and widely adopted in engineering simulations, can become computationally expensive, particularly when applied to large-scale multibody systems with strong nonlinearities and tightly coupled aeroelastic interactions [4][3].

The choice of Cp-Lambda as a case study is motivated by several factors. First, it represents a realistic and industrially relevant legacy HPC application, originally developed for x86 CPU architectures and reflective of software design practices common in the 2000–2010 period. Second, its structure combines computationally intensive finite element operations with iterative nonlinear solvers, making it an ideal candidate for performance analysis and optimization. Finally, the presence of coupled C and MATLAB components, along with its modular but non-fully modernized architecture, provides a meaningful testbed to investigate how legacy scientific codes can be adapted to modern parallel computing environments.

2 | Problem Description

2.1. State of the Art of the Original Solver

This section describes the original multibody finite element codebase as it was initially proposed, before the introduction of modern numerical libraries and performance-oriented modifications. The discussion focuses on the dynamic finite element workflow, the custom sparse linear solver, and the main software-engineering characteristics of the implementation. The aim is to assess the original design in terms of numerical structure, modeling capabilities, and computational limitations.

2.1.1. General Architecture

The original code is organized as a procedural C framework for nonlinear structural and aeroelastic simulation. Its execution flow combines several responsibilities within a single simulation environment: finite element assembly, time integration, nonlinear iteration, solution of sparse linear systems, aerodynamic coupling, controller and sensor management, restart handling, and archival output.

At a high level, the solver proceeds through the following stages:

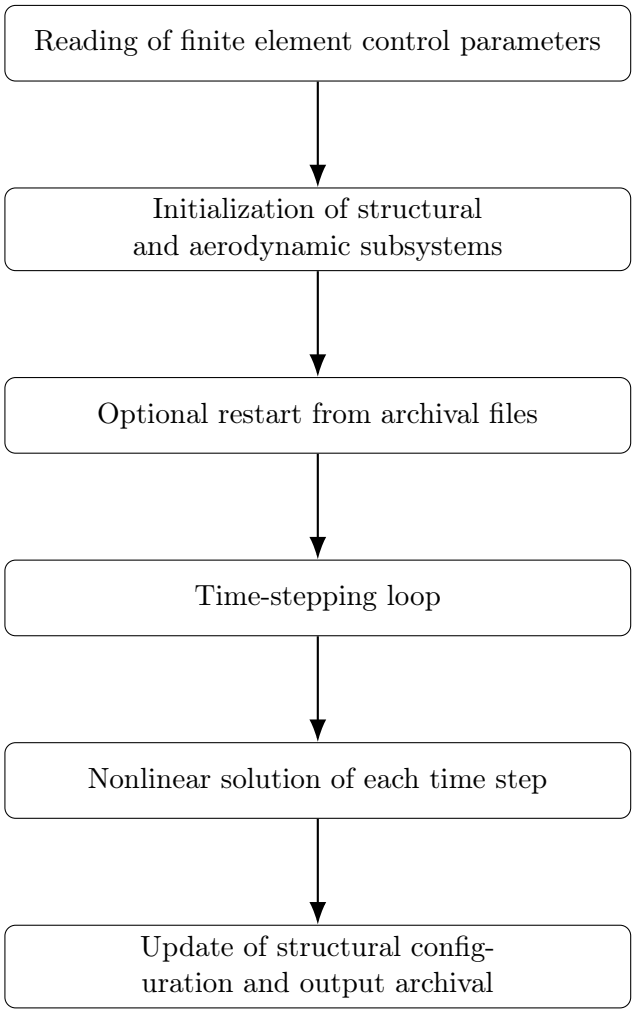

Figure 2.1: High-level workflow of the original Cp-Lambda solver

Dynamic Solution Strategy The dynamic analysis is based on an incremental nonlinear procedure consistent with a Newton–Raphson framework. For each time step, the code performs repeated iterations until either convergence is achieved or the maximum number of iterations is reached. Each nonlinear iteration includes:

1. definition of the matrix factorization policy;
2. assembly of structural matrices and equivalent force terms;
3. evaluation of external and aerodynamic loads;
4. solution of the linearized system;
5. update of displacement increments;
6. evaluation of convergence indicators.

The solver supports several convergence measures, including displacement-based, force-based, and energy-based criteria. This is a relevant strength of the original implementa-

tion, since it shows numerical awareness and the ability to adapt convergence monitoring to different classes of nonlinear problems.

Time-Stepping and Adaptivity The code includes a built-in adaptive time-stepping logic. The time-step size may be modified according to three main conditions:

- lack of nonlinear convergence;
- detection of event-related conditions, such as contact or discontinuity activation;
- estimated local discretization error.

This is an important feature for nonlinear dynamics and aeroelastic simulations, where fixed-step schemes are often inadequate. The presence of rejection and retry mechanisms indicates that the original code was designed with robustness in mind, especially for strongly nonlinear transient analyses.

Structural Assembly and Modeling Scope One of the most significant strengths of the software is the breadth of its physical modeling. The global structural system is assembled by combining contributions from many different element and joint families. The matrix sparsity pattern is precomputed once from the model topology and then reused during the simulation. This reduces repeated structural overhead and reflects a coherent finite element implementation strategy.

Software Characteristics From a software-engineering viewpoint, the code follows a typical legacy scientific-programming style:

- strong reliance on procedural C;
- extensive use of global state and shared object registries;
- direct data access through macros and simple structures;
- coupling between numerical kernels, control logic, and I/O;
- platform assumptions consistent with historical x86 and Windows-oriented development.

This style was common and effective for research and industrial codes developed in the early 2000s. It favors direct control over memory and execution flow, but it also reduces modularity, maintainability, portability, and ease of modernization.

Custom Sparse Linear Solver The global linear system is handled through a custom direct solver based on skyline storage. The solver includes:

- compact storage of the global stiffness matrix;
- separate treatment of symmetric and unsymmetric cases;
- in-house factorization routines;
- forward and backward substitution procedures.

For the original target environment, this was a reasonable engineering choice. Skyline storage reduces memory usage compared with dense storage and fits naturally with sequential finite element assembly. Moreover, a direct solver provides deterministic behavior and avoids the convergence uncertainty of iterative linear methods.

However, from a contemporary HPC perspective, this component is also the most evident numerical bottleneck. The storage format is rigid, the factorization kernels are inherently sequential, and the implementation lacks the advanced reordering, pivoting, and scalability features available in modern sparse direct solvers.

2.2. Analyzing Strengths and Weaknesses

The original Cp-Lambda solver exhibits a combination of strong numerical capabilities and clear architectural limitations, which are typical of legacy scientific software developed in the early 2000s. Its design reflects a pragmatic engineering approach, prioritizing functionality and robustness over modularity and computational efficiency from a modern HPC perspective.

Main Strengths of the Solver

- **Numerical workflow** The solver is based on a clear nonlinear incremental-iterative strategy for both static and dynamic analyses. This provides a solid and well-understood numerical foundation, ensuring consistency in the treatment of nonlinear structural response.
- **Convergence control** Multiple convergence criteria are implemented, together with step rejection mechanisms. This contributes significantly to the robustness of the nonlinear solution process, particularly in challenging aeroelastic configurations.
- **Adaptive time integration** The presence of adaptive time stepping allows the solver to adjust the computational step size based on convergence behavior, improving both stability and efficiency in transient analyses.
- **Modeling breadth** The code supports a wide range of structural elements, joints, and aeroelastic components, making it highly versatile for multibody simulation scenarios.
- **Sparse structure handling** The use of a precomputed skyline storage structure reduces overhead during repeated assembly operations and reflects an efficient strategy for the types of problems originally targeted by the code.
- **Deterministic solver behavior** The use of direct factorization methods ensures reproducibility of results, which is particularly important in engineering validation contexts.

Main Weaknesses of the Solver

- **Software coupling** The codebase is highly monolithic, with solver logic, physical modeling, control flow, and I/O operations tightly interconnected. This reduces maintainability and complicates any attempt at modular refactoring.
- **Global-state dependence** The extensive use of shared global structures limits encapsulation and makes the software difficult to extend or parallelize safely.
- **Sparse linear algebra approach** The custom skyline-based solver, while efficient in its original context, is less flexible and less scalable than modern sparse matrix libraries based on CSR/CSC formats.
- **Memory behavior** Data access patterns are not optimized for modern cache hierarchies, leading to suboptimal performance on current CPU architectures.
- **Lack of parallelism** The core computational kernels are fundamentally sequential and do not exploit multicore or manycore architectures.
- **Branch-heavy execution** Frequent conditional logic within performance-critical loops limits vectorization and reduces pipeline efficiency.
- **I/O overhead** Extensive formatted output inside iterative solution loops introduces unnecessary runtime overhead.
- **Limited portability** The implementation reflects assumptions typical of older x86 and Windows-centered execution environments, reducing portability across modern HPC systems.

Considarations about the Main Limitations

- **Numerical Linear Algebra Limitations**

The custom skyline solver is coherent with the historical context of the code, but it is not ideal for present-day large-scale simulation. Skyline storage is efficient only when the matrix profile remains sufficiently regular. In more complex multibody and aeroelastic problems, it becomes less competitive than compressed sparse formats used by modern libraries. In addition, the custom factorization routines expose limited numerical safeguards compared with state-of-the-art sparse direct solvers.

- **Performance Limitations**

The code was clearly conceived for sequential x86 execution. Its core operations combine scalar loops, pointer-based traversals, conditional branching, and repeated formatted output. Such patterns were acceptable in the original target environment, but they interact poorly with current cache hierarchies, vector units, and thread-level parallelism.

- **Software-Engineering Limitations**

The implementation is rich in functionality but not strongly modular. Numerical kernels are embedded in a broader procedural control flow that includes restart logic, output

management, and subsystem orchestration. As a result, the code is effective as a monolithic simulation application, but less suitable as a flexible computational framework for modern solver integration.

2.3. Code Profiling

The first step of the optimization process consisted in performing an extensive profiling analysis of the application. The objective of this initial investigation was to identify the main computational bottlenecks, evaluate the distribution of execution time among the different software components, and obtain a preliminary understanding of the code behavior in terms of CPU usage, memory access, and input/output operations.

Table 2.1: Summary of the first profiling analysis

Metric / Function	**Value**	**Contribution**
Total CPU time in `Time Analysis FEM`	40599	97.20%
CPU time in `Dynamic Solution`	39976	95.71%
CPU time in `Linear System Solution`	23689	56.72%
CPU time in `Unsymmetric Factorization`	12124	29.03%
CPU time in `Unsymmetric Solution`	11508	27.55%
CPU time in `Dynamic Compute Structural Matrices`	8298	19.87%
CPU time in `Dynamic Applied Loads`	7914	18.95%
File read operations (1000 time steps)	∼10000	>36 MB read
File write operations (1000 time steps)	∼8000	∼32 MB written
File read operations (100 time steps)	∼10000	>36 MB read
File write operations (100 time steps)	∼200	∼7 MB written

A first observation emerged from the analysis of the input/output activity. Even on relatively small simulations, the amount of file operations appeared significant. Considering a preliminary execution of 1000 time steps, while a complete production simulation generally exceeds 50000 time steps, the application already performs nearly 10000 file read operations, corresponding to more than 36 MB of data read, together with approximately 8000 write operations producing around 32 MB of output data.

Interestingly, when reducing the simulation length to only 100 time steps, the number of read operations remains almost unchanged, whereas write operations decrease substantially to slightly less than 200, with approximately 7 MB written. This behavior suggests that a relevant portion of the input data is loaded during initialization and remains mostly independent from the simulation duration, while output generation scales more directly with the number of simulated time steps. Such characteristics indicate that the initialization phase may introduce a negligible overhead, but we need to pay more attention to the extensive use of write-on-file operations.

The profiling analysis also highlighted the computational dominance of a limited set of numerical routines. The overall execution flow is almost entirely concentrated inside the

`Time Analysis FEM` routine, which accounts for approximately 97% of the total CPU time. Within this function, the `Dynamic Solution` procedure alone occupies more than 95% of the total execution time, immediately identifying the nonlinear dynamic solver as the core computational hotspot of the application.

A more detailed inspection reveals that the most expensive operations are associated with the solution of the linear systems generated during the Newton–Raphson iterations. In particular, the routine `Linear System Solution` accounts for approximately 57% of the total CPU time. Inside this component, two functions dominate the computational cost:

- `Unsymmetric Factorization` with approximately 29% of the total execution time;
- `Unsymmetric Solution` with approximately 27% of the total execution time.

These results clearly indicate that matrix factorization and linear system solution routines represent the primary computational bottleneck of the entire application. This is consistent with the expected behavior of nonlinear finite element solvers, where repeated assembly and solution of large sparse systems typically dominate the execution cost.

Additional relevant contributions are associated with the computation of structural matrices and applied loads. The function `Compute Structural Matrices` accounts for nearly 20% of the total CPU time, while `Applied Loads` contributes approximately 19%. These routines involve intensive floating-point computations and repeated memory accesses, making them potential candidates for further optimization and parallelization strategies.

Operations related to convergence checks, displacement updates, controller management, and logging exhibit negligible computational weight compared to the dominant solver routines. This observation is particularly important because it allows the optimization effort to remain focused on a relatively small subset of critical kernels instead of dispersing resources across the entire codebase.

Overall, the first profiling phase provided a clear overview of the software behavior and identified the main areas where optimization efforts could produce meaningful performance improvements. The results suggest that the modernization process should primarily focus on:

- improving the efficiency of matrix factorization and linear system solution routines;
- reducing memory access inefficiencies during structural matrix assembly;
- investigating opportunities for parallelization of the most computationally intensive kernels;
- evaluating possible reductions of unnecessary input/output operations.

These observations constituted the foundation for the subsequent optimization strategies adopted during the development of this work.

2.4. Sparse Representation and Structure of the Global Stiffness Matrix

The global stiffness matrix arising from the finite element formulation of the problem is stored using a **skyline (profile) sparse storage format**. This representation is a classical approach widely adopted in legacy finite element codes, particularly when memory efficiency and direct solvers are preferred over iterative methods.

In a skyline storage scheme, the structure is typically decomposed into three main components:

- **Diagonal vector**: stores the diagonal entries of the stiffness matrix;
- **Skyline (profile) vector**: stores the offsets that define the first non-zero entry in each column, effectively describing the bandwidth structure;
- **K matrix (value array)**: stores all coefficients within the skyline profile in a compact 1D array.

$$
K = \begin{bmatrix} 10 & -2 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 \\ 0 & -3 & 15 & 0 & -1 \\ 0 & 0 & 0 & 8 & -2 \\ 0 & 0 & -1 & -2 & 11 \end{bmatrix}
$$

$$
\begin{aligned}
&\texttt{SKYLINE}: \\
&\texttt{skyline} = [1, 1, 2, 4, 3] \\
&\texttt{diagonal} = [1, 7, 12, 15, 19] \\
&\texttt{values} = [10, -2, 0, 0, 0, -2, 12, -3, 0, 0, -3, 15, 0, -1, 8, -2, -1, -2, 11] \\
&\texttt{CSR}: \\
&\texttt{col\_idx} = [1, 2, 1, 2, 3, 2, 3, 5, 4, 5, 3, 4, 5] \\
&\texttt{row\_ptr} = [1, 3, 6, 8, 9, 11] \\
&\texttt{values} = [10, -2, -2, 12, -3, -3, 15, -1, 8, -2, -1, -2, 11] \\
&\texttt{COORDINATE}: \\
&\texttt{col\_idx} = [1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5] \\
&\texttt{row\_idx} = [1, 2, 1, 2, 3, 2, 3, 5, 4, 5, 3, 4, 5] \\
&\texttt{values} = [10, -2, -2, 12, -3, -3, 15, -1, 8, -2, -1, -2, 11]
\end{aligned}
$$

(2.1)

Equation 2.1 shows an example of a sparse matrix stored in three different formats, the already descripted `skyline`, `CSR` and `COORDINATE` format.

Skyline in particular significantly reduces memory consumption compared to full matrix storage, especially when the matrix exhibits a narrow bandwidth structure, as is typical in finite element discretizations of structural systems. However, its efficiency strongly depends on the ordering of degrees of freedom and the sparsity pattern, and still stores useless 0 values in comparison to other formats. Another limitation of this storage is that is less flexible than modern compressed sparse formats and can lead to suboptimal performance on parallel architectures due to indirect memory access patterns and limited vectorization opportunities.

2.4.1. Case Study: Matrix Characteristics

For the considered wind turbine model, consisting of **4362 degrees of freedom (DOFs)**, the stiffness matrix exhibits the following characteristics:

Table 2.2: Skyline storage structure and matrix characteristics

Component / Metric	**Value**	**Memory / Notes**
Degrees of freedom (DOF)	4362	-
Diagonal vector	4362 int	∼17 KiB
Skyline vector	4362 int	∼17 KiB
K matrix (stored values)	432,724 double	∼3.3 MiB
Non-zero entries	129,813	∼30% of stored values
Effective stored sparsity	∼2% of full matrix	highly structured bandwidth

This corresponds to a sparsity level where only a small fraction of the full matrix is explicitly stored. In particular, only about **2%** of the full matrix entries are stored, and within the stored profile only about **30%** of values are non-zero, highlighting a strongly structured but still partially sparse pattern.

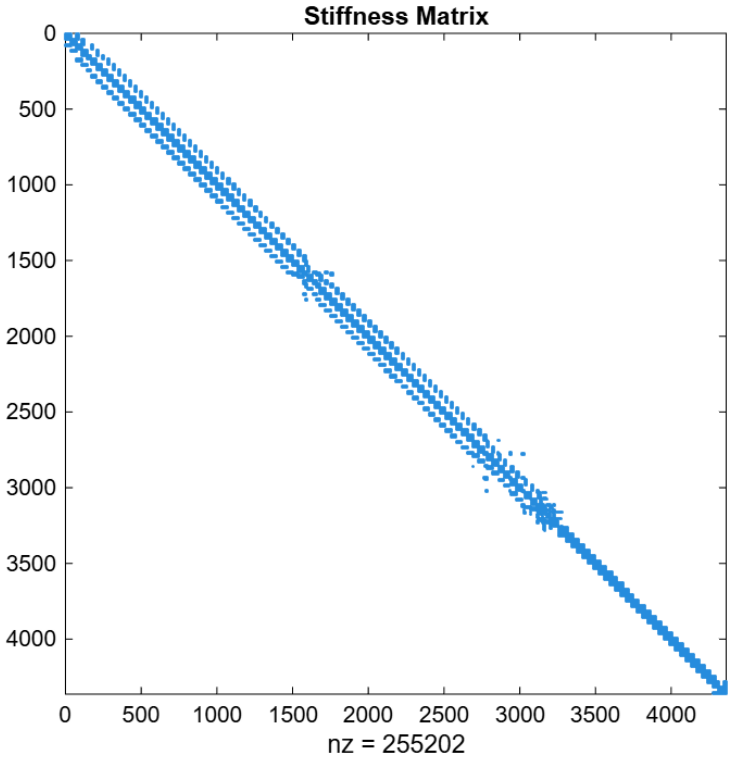

Figure 2.2: Sparsity pattern of the global stiffness matrix (skyline representation)

The observed structure confirms that the stiffness matrix is strongly banded, with a relatively narrow bandwidth and a sparsity pattern consistent with classical finite element discretizations of multibody structural systems. The skyline format, while not the most modern sparse storage scheme, is well suited to such problems due to its deterministic memory layout and compatibility with direct solvers.

Given this structure, the use of a custom LU factorization algorithm tailored to skyline storage appears to have been a reasonable design choice at the time of development. Skyline-based LU factorization inherently limits fill-in to the predefined profile, ensuring predictable memory usage and avoiding uncontrolled growth of non-zero entries during factorization.

To better address the problem of matrix allocation Table 2.3, shows how the different matrix formats impact the memory allocation and the effective number of entries:

Table 2.3: Memory comparison between dense and sparse storage formats for the same stiffness matrix ($n = 4362$, $nnz = 255256$).

Format	Memory footprint	Memory [MB]	Storage ratio vs dense
Dense matrix	4362^2 doubles	145.16	100%
Skyline	432724 doubles + 8724 ints	3.41	2.27%
CSR	255256 doubles + 255256 ints + 4363 ints	4.13	1.34%
COO	255256 doubles + 2×255256 ints	5.85	1.34%

The results clearly show that the Skyline representation achieves the lowest memory footprint for the considered problem. However, this advantage is strongly problem-dependent: for more irregular sparsity patterns or larger-scale systems, the storage efficiency of Skyline degrades rapidly, whereas CSR/COO formats maintain memory requirements proportional only to the actual number of non-zero entries, making them significantly more robust and scalable for general-purpose sparse solvers.

2.5. Conclusions

The original code should be regarded as a solid legacy engineering solver: numerically meaningful, physically rich, and operationally complete. Its main strengths lie in the reliability of its simulation workflow, the breadth of supported models, and the presence of convergence and adaptivity mechanisms. Its main weaknesses lie in the monolithic software structure, the custom skyline-based sparse solver, and an implementation style that reflects the assumptions of an earlier x86-oriented computing era. For this reason, the code represents a valuable starting point for modernization, even though its original design is not naturally aligned with contemporary HPC practices.

3 | Adopted strategies

This chapter presents the set of optimization strategies adopted in order to improve the performance and efficiency of the legacy Cp-Lambda code. Based on the analysis of the original implementation, the most relevant computational bottlenecks have been identified in numerical algorithms, memory usage, and execution flow. Based on this assessment, targeted modifications have been introduced.

Rather than redesigning the entire application from scratch, the approach follows an incremental modernization strategy. This choice preserves the original numerical models and the solver's physical consistency, while enabling systematic improvements in computational efficiency and scalability. The focus is placed on the components that have the highest impact on runtime performance.

3.1. Codebase Modernization and Platform Migration

Before introducing direct optimization strategies, the first major step of this work consisted in modernizing the original codebase in order to support both **x64** and subsequently **UNIX/Linux** compilation environments. Although this phase was not initially conceived as a performance optimization activity, it rapidly became one of the most time-consuming and technically challenging parts of the entire project.

The motivation behind this migration process was twofold. First, modern 64-bit compilation environments provide access to more advanced compiler optimizations and more aggressive optimization flags compared to legacy x86 targets. In addition, contemporary UNIX-based HPC ecosystems offer significantly better support for modern scientific computing libraries, particularly for sparse linear algebra, numerical solvers, and parallel computing frameworks. This aspect is especially important for applications involving large finite element systems and computationally intensive matrix operations.

A second motivation concerned software portability and long-term maintainability. Modern HPC infrastructures are predominantly Linux-based and heavily rely on toolchains optimized for heterogeneous architectures and accelerator-oriented programming models. Consequently, maintaining compatibility with legacy x86-only environments would strongly limit future scalability and integration possibilities.

Despite appearing conceptually straightforward, the migration process revealed numerous hidden issues within the original codebase. The primary challenge was not simply

obtaining a successful compilation, but rather achieving numerical results consistent with the original implementation. Several bugs and unsafe programming practices remained effectively hidden in the x86 environment due to differences in memory layout, compiler behavior, and runtime protections.

The migration therefore required a parallel effort involving both the update of the core source code and the adaptation of the external dynamic libraries (`.dll`) on which the application depended. In many cases, behavior that appeared correct under x86 execution produced crashes, numerical instabilities, or invalid results once compiled in x64 or UNIX environments.

After extensive debugging and validation, a list of the most common and critical issues was identified. These problems are particularly relevant for anyone attempting to modernize legacy scientific software originally developed for older architectures.

Table 3.1: Most common issues encountered during x64 and UNIX migration

Bug / Cause	**Observed Error**	**x64**	**UNIX**
Pointer addresses stored inside `int` variables or arrays, despite 64-bit addresses requiring 8 bytes	Segmentation fault caused by truncated memory addresses	✓	
Read/write operations performed outside allocated array boundaries	Segmentation fault due to stricter memory protection	✓	
Variables declared but not initialized before accumulation operations (`+=`)	Non-convergent numerical model or non-physical results caused by random initial values	✓	
Compatibility issues in external library management and dynamic linking	Crash when accessing external libraries		✓
Errors in file path parsing and path management	Crash due to missing files or invalid paths		✓

This phase highlighted an important aspect of legacy scientific software modernization: older architectures and compilers often tolerate unsafe memory operations or undefined behaviors that become critical once moving toward modern systems. As a consequence, the migration process itself became an implicit debugging and validation stage, revealing structural weaknesses that had remained hidden for years.

3.2. General Optimizations

After completing the first profiling phase and the codebase modernization process, a set of preliminary optimization strategies was introduced based on a high-level inspection of the application structure. These optimizations were designed to reduce unnecessary

overhead in the most frequently executed portions of the code before attempting more invasive algorithmic modifications.

The initial analysis revealed that several deeply nested loops contained operations that were computationally inefficient or poorly suited for modern processor architectures. In particular, the following recurring issues were identified:

- Conditional branches executed inside highly iterative loops;
- Frequent function calls used only to retrieve constant or slowly-varying parameters;
- Repeated input/output operations performed directly inside computational kernels.

Although individually inexpensive, these operations become extremely costly when executed millions of times during long transient simulations.

3.2.1. Loop Simplification and Branch Reduction

A first optimization step consisted in simplifying the innermost computational loops. Several conditional branches originally executed at every iteration were moved outside the loops whenever possible. This approach reduces branch misprediction penalties and improves instruction pipeline efficiency.

Similarly, numerous function calls used only to access configuration parameters or retrieve values that remained constant during the loop execution were removed from the iterative regions. These quantities were instead computed or loaded once before entering the loop and subsequently reused locally.

The main objectives of these modifications were:

- reducing unnecessary instruction overhead;
- improving cache locality and instruction predictability;
- facilitating compiler vectorization and optimization;
- simplifying the computational kernels before parallelization attempts.

3.2.2. Buffered File Output

Particular attention was dedicated to the management of file output operations. The profiling phase had already highlighted a large number of write operations during simulations, especially for long transient analyses.

Originally, the code relied heavily on direct `fprintf()` calls performed repeatedly inside iterative procedures. While each write operation involved relatively small amounts of data, the cumulative overhead associated with frequent filesystem access became significant.

To mitigate this issue, a custom buffered output mechanism named `buffered_fprintf()` was introduced. Instead of writing directly to disk at every call, this function accumulates formatted output data inside an internal memory buffer of configurable size. The actual file write operation is performed only when:

- the buffer becomes full;
- an explicit flush operation is requested.

The introduction of buffered file writing did not reduce the total amount of written data, which remained approximately unchanged, but it produced a substantial reduction in the number of write operations performed during execution.

The reduction of filesystem accesses resulted in:

- lower I/O overhead;
- improved execution continuity inside computational kernels;
- reduced synchronization and operating system interaction costs;
- better scalability for long simulations.

This optimization proved particularly beneficial considering that complete production simulations may involve tens of thousands of time steps and therefore generate extremely large numbers of output operations if handled naively.

The modifications introduced during this phase aimed to improve the overall efficiency of the application while keeping the original numerical formulation unchanged, thus preserving the validity and reproducibility of the simulation results.

3.3. Solver Analysis and Numerical Solution Strategies

3.3.1. Initial Considerations on Solver Selection

One of the most critical aspects of the optimization process concerned the numerical solution of the linear systems arising during the Newton–Raphson iterations. As highlighted by the profiling analysis, matrix factorization and linear system solution routines account for the majority of the total execution time, making the solver architecture the primary target for performance improvements.

The problem considered in this work involves approximately **4400 degrees of freedom (DOFs)**. Although this size is significant from a computational perspective, it cannot be classified as a very large-scale sparse problem by modern HPC standards. This observation immediately influences the choice of the most appropriate numerical solution strategy.

In addition, the previous analysis of the global stiffness matrix structure revealed several important characteristics:

- the matrix is sparse;
- the matrix is non-symmetric;
- the sparsity pattern exhibits a relatively narrow diagonal bandwidth;
- the skyline storage format already limits fill-in growth during factorization.

These properties strongly suggest the adoption of a **direct solver approach** rather than an iterative one. For matrices with narrow bandwidth and moderate size, direct factorization methods can often outperform iterative solvers due to their predictable convergence behavior and reduced iteration overhead.

From a computational complexity perspective, the difference becomes particularly relevant. For a generic sparse matrix of size n:

- iterative methods typically require repeated sparse matrix-vector products with computational cost approximately proportional to:

  $$O(k \cdot nnz)$$

  where k is the number of iterations and nnz is the number of non-zero entries;
- direct LU factorization for banded matrices with bandwidth b has computational complexity approximately proportional to:

  $$O(n \cdot b^2)[9]$$

  while forward and backward substitutions scale as:

  $$O(n \cdot b)$$

Since the considered stiffness matrix exhibits a relatively small bandwidth, the effective computational cost of the direct approach remains manageable and highly competitive. Furthermore, the deterministic nature of direct solvers eliminates convergence uncertainties typically associated with iterative methods.

These considerations also have important implications regarding GPU acceleration. Modern GPU architectures are highly optimized for massively parallel workloads and therefore tend to favor iterative solvers based on sparse matrix-vector operations. However, in the present case several factors make GPU acceleration less attractive:

- the problem size is relatively moderate;
- the matrix bandwidth is narrow;
- direct factorization already limits fill-in efficiently;
- iterative methods would introduce additional convergence overhead;
- sparse iterative GPU solvers require repeated transfers and synchronization across memory hierarchies.

Moreover, the use of iterative methods on this type of problem may lead to additional fill-in and preconditioning costs that can easily offset the benefits of GPU parallelism. As a consequence, preliminary analysis suggested that a well-optimized CPU direct solver would likely outperform GPU-oriented iterative approaches for this specific application[14].

The original custom solver already implements a direct LU factorization strategy without pivoting, followed by classical forward and backward substitution procedures. While relatively simple, this implementation is naturally compatible with the skyline matrix structure and benefits from the narrow bandwidth characteristics of the problem.

Based on these observations, the optimization strategy focused not on replacing the direct approach itself, but rather on evaluating whether modern highly optimized external sparse direct solvers could provide better performance than the original implementation.

Two external solver libraries were therefore selected for comparison:

- **MUMPS** (MUltifrontal Massively Parallel Sparse Solver);
- **Intel MKL PARDISO**.

Both libraries are widely adopted in scientific computing and provide highly optimized sparse direct factorization algorithms capable of exploiting modern multicore architectures, advanced memory management techniques, and efficient reordering strategies.

3.3.2. MUMPS Sparse Direct Solver

The first external solver considered for the performance comparison is **MUMPS (MUltifrontal Massively Parallel Sparse direct Solver)**, a widely used library for the solution of large sparse linear systems on distributed and shared-memory architectures. MUMPS implements a **multifrontal direct method** based on Gaussian elimination, designed to efficiently handle sparse matrices arising from finite element discretizations and similar engineering applications.

General characteristics and input format

MUMPS accepts matrices in several sparse formats, the most common being the **coordinate (COO) format**, where the matrix is defined as a list of triplets (i, j, a_{ij}). This format is particularly convenient for finite element assembly procedures, since elemental contributions can be directly accumulated without requiring a predefined storage structure.

Internally, MUMPS converts the input matrix into more efficient compressed representations during preprocessing. The solver is designed to operate on:

- general unsymmetric matrices;
- symmetric positive definite matrices;
- general symmetric matrices.

Three-phase execution workflow

The computational workflow of MUMPS is divided into three main phases[13]:

1. **Analysis phase**

2. **Factorization phase**
3. **Solution phase**

The **analysis phase** is particularly important, as it performs:

- reordering of the matrix (typically using algorithms such as METIS, SCOTCH, or AMD);
- symbolic factorization;
- construction of the **elimination tree**, which describes dependencies between variables;
- estimation of memory requirements and computational cost.

The symbolic factorization does not involve numerical values; instead, it analyzes the sparsity pattern of the matrix in order to predict the structure of the LU (or LDL^T) factors and to minimize fill-in during the subsequent numerical factorization.

During the **numerical factorization phase**, MUMPS performs a multifrontal LU decomposition:

$$A = LU$$

or, in the symmetric case,

$$A = LDL^T.$$

The method operates by assembling *frontal matrices*, which are small dense blocks associated with nodes of the elimination tree. Each frontal matrix is factorized using optimized dense linear algebra kernels (BLAS/LAPACK), while contributions are propagated upward through the tree.

Key aspects that make MUMPS highly efficient include:

- **multifrontal decomposition**, which reduces sparse factorization to a sequence of dense operations;
- **dynamic scheduling**, allowing better load balancing across processors;
- extensive use of optimized BLAS routines for dense kernels;
- advanced ordering techniques that reduce fill-in;
- support for parallel execution (MPI + OpenMP hybrid model).

These characteristics make MUMPS particularly efficient for large-scale sparse problems, especially when memory and computational costs are dominated by factorization.

Software environment and limitations

MUMPS is primarily designed for **UNIX/Linux-based HPC systems** and is tightly integrated with MPI-based parallel environments and optimized numerical libraries such as BLAS and ScaLAPACK. As a consequence, it is not natively available in typical Windows/x86 legacy environments without significant adaptation layers.

The main strengths of MUMPS can be summarized as follows:

- excellent robustness for general sparse systems;
- high performance due to multifrontal approach and dense kernel optimization;
- good scalability on multicore and distributed systems;
- effective fill-in reduction through advanced ordering strategies;
- wide adoption in scientific and industrial HPC applications.

However, some limitations are also present:

- relatively high memory consumption due to fill-in in frontal matrices;
- complexity of configuration and tuning for optimal performance;
- strong dependency on UNIX-like environments;
- performance sensitivity to matrix ordering and problem size;
- less efficient for small to medium-sized problems compared to simpler direct solvers.

In the context of this work, MUMPS represents a state-of-the-art reference for sparse direct solvers and provides a meaningful benchmark for evaluating the performance of the original custom skyline-based LU solver. Its combination of symbolic analysis, multifrontal factorization, and parallel execution makes it particularly suitable for modern HPC architectures, although its full benefits are typically observed on larger-scale problems than the one considered in this study.

3.3.3. Intel MKL PARDISO Sparse Direct Solver

The second external numerical solver considered in this work is **Intel MKL PARDISO**. Similarly to MUMPS, PARDISO is a state-of-the-art sparse direct solver designed for the efficient solution of large sparse linear systems, but it is primarily optimized for **shared-memory architectures** and tightly integrated within the Intel Math Kernel Library (MKL) ecosystem[11].

While both solvers belong to the class of multifrontal or supernodal direct methods, PARDISO distinguishes itself through a strong focus on cache efficiency, supernode compression, and optimized use of multicore CPUs. For the problem under consideration, this makes it a natural candidate for comparison with both the custom skyline-based LU solver and the previously analyzed MUMPS implementation.

General characteristics and input format

In contrast to MUMPS, which typically operates starting from a coordinate (COO) representation, PARDISO requires the input matrix to be provided in a **compressed sparse format**, most commonly **CSR (Compressed Sparse Row)**.

The CSR format consists of:

- a value array containing all non-zero entries;
- a column index array;
- a row pointer array defining row boundaries.

This structure is particularly efficient for memory access patterns and cache utilization, making it well suited for high-performance implementations on modern CPU architectures.

PARDISO is designed to handle:

- general unsymmetric matrices;
- symmetric positive definite matrices;
- symmetric indefinite matrices.

Multiphase solver workflow

Similarly to MUMPS, the PARDISO solver is organized into distinct computational phases:

1. **Analysis phase**
2. **Factorization phase**
3. **Solution phase**
4. **Iterative refinement (optional)**

The **analysis phase** plays a crucial role in the overall performance of the solver. It performs operations analogous to those already discussed for MUMPS, including matrix reordering and symbolic factorization. The goal is to reduce fill-in during numerical factorization and to determine an efficient elimination ordering.

PARDISO typically relies on advanced graph partitioning and ordering techniques such as nested dissection (often via METIS), combined with minimum degree heuristics for local refinement.

The symbolic factorization step determines the structure of the LU or LDL^T factors without performing numerical computations, enabling efficient memory allocation and minimizing dynamic memory overhead during factorization.

During the **numerical factorization phase**, PARDISO performs a sparse direct factorization based on a **supernode-based LU/LDL^T** decomposition:

- $A = LU$ for general unsymmetric systems;
- $A = LDL^T$ for symmetric or symmetric indefinite systems.

Compared to the multifrontal approach used by the solver previously described (MUMPS), PARDISO relies more heavily on **supernodes**, which group columns with similar sparsity

patterns into dense blocks. These blocks are then processed using highly optimized BLAS Level 3 (matrix-matrix operations) routines, maximizing computational efficiency.

Key performance-oriented characteristics include:

- aggressive exploitation of cache hierarchies;
- reduced overhead due to supernode compression;
- extensive use of vectorized BLAS kernels;
- efficient parallelization through OpenMP threading;
- optional iterative refinement to improve numerical accuracy.

These features make PARDISO particularly efficient for medium-sized to large sparse systems where memory locality and dense kernel performance dominate overall execution time.

Software environment and limitations

Unlike MUMPS, which is designed for distributed-memory HPC environments, PARDISO is primarily optimized for **shared-memory systems**. It is distributed as part of the Intel MKL library and is therefore mainly available and optimized on **UNIX/Linux systems** with Intel compilers and runtime environments.

Although it can be used on other platforms, its best performance is typically achieved in tightly controlled HPC environments where Intel architectures and MKL optimizations are fully exploited.

The main strengths of PARDISO include:

- extremely high performance on shared-memory multicore architectures;
- efficient memory usage due to supernode compression;
- strong cache and vectorization optimization;
- robust numerical stability with iterative refinement;
- straightforward integration within the MKL ecosystem.

However, some limitations can also be identified:

- limited scalability in distributed-memory environments compared to solvers such as MUMPS;
- strong dependency on Intel MKL and related ecosystem;
- reduced flexibility compared to more general-purpose sparse frameworks;
- performance sensitivity to ordering and sparsity structure, similarly to other direct solvers already discussed.

In continuity with the previously analyzed MUMPS solver, Intel MKL PARDISO provides a complementary perspective on sparse direct solution strategies. While MUMPS is more oriented toward distributed and hybrid parallel environments, PARDISO focuses on highly optimized shared-memory execution.

Table 3.2: Comparison between MUMPS, PARDISO and the custom skyline LU solver

Feature	**MUMPS**	**Intel MKL PARDISO**	**Custom Solver (Skyline LU)**
Matrix format	COO (coordinate format), converted internally to compressed structures	CSR (Compressed Sparse Row)	Skyline (profile storage with diagonal + skyline + value array)
Factorization strategy	Multifrontal LU/LDL^T decomposition	Supernode-based LU/LDL^T factorization	Classical LU factorization without pivoting on skyline structure
Symbolic analysis	Yes (ordering + elimination tree + fill-in prediction)	Yes (graph-based ordering + symbolic factorization)	No explicit symbolic analysis, structure is fixed by skyline profile
Pivoting strategy	Partial pivoting (robust numerical stability)	Partial pivoting + iterative refinement	No pivoting (diagonal must remain nonzero or regularized manually)
Parallelization	MPI + OpenMP hybrid (distributed + shared memory)	OpenMP (shared memory optimized)	None (serial implementation)
Computational kernel	Dense frontal matrices solved via BLAS/LAPACK	Dense supernode kernels using BLAS Level 3	Explicit skyline loops with manual elimination and substitution
Memory strategy	Dynamic memory with significant fill-in control via ordering	Highly optimized cache-aware memory layout	Fixed skyline storage, no dynamic reordering or compression changes
Strengths	Very robust, scalable, excellent for large sparse systems	Extremely fast on shared-memory systems, cache optimized	Very simple, deterministic, efficient for small/moderate banded systems
Weaknesses	High memory consumption, complex setup	Limited distributed scalability, Intel-dependent	No pivoting (numerical risk), no parallelism, poor scalability for large systems
Typical complexity behavior	Depends on fill-in (close to supernodal LU complexity)	Optimized close to $O(n \cdot b)$–$O(n \cdot b^2)$ depending on sparsity	Approximately $O(n \cdot b^2)$ for factorization, $O(n \cdot b)$ for substitution
Target architecture	HPC clusters (distributed + shared memory)	Multicore shared-memory CPUs	Legacy x86 serial CPU architectures

3.3.4. Limitations of the Custom Skyline LU Solver in Modern HPC Contexts

Although the custom solver is based on a conceptually simple LU factorization over a skyline storage structure, its performance and scalability limitations become evident when compared to modern sparse direct solvers. In particular, two key aspects explain why this approach increasingly represents a computational bottleneck and is no longer competitive in contemporary HPC environments.

Why the custom solver becomes a bottleneck

Despite its algorithmic simplicity, the custom solver exhibits several structural inefficiencies that limit its performance:

- **Serial execution**: the implementation is fully sequential, with no exploitation of multicore architectures or parallel linear algebra kernels.
- **Lack of optimized kernels**: all operations are performed through explicit nested loops, without relying on highly optimized BLAS Level 3 routines.
- **Memory access patterns**: skyline traversal introduces indirect and non-contiguous memory accesses, reducing cache efficiency.
- **Repeated arithmetic overhead**: factorization and substitution phases rely on scalar operations inside deeply nested loops.

As a consequence, even for moderate problem sizes (e.g., $\sim$4000 DOFs), the solver becomes the dominant cost of the simulation once higher-level optimizations are applied elsewhere in the code.

Why skyline LU is no longer competitive in modern HPC environments

The skyline LU approach, while historically effective for banded finite element problems, shows clear limitations in the context of modern HPC architectures:

- **No symbolic analysis**: the sparsity pattern is fixed a priori, preventing reordering strategies that reduce fill-in.
- **No pivoting strategy**: numerical stability is not guaranteed in general cases, requiring problem-specific assumptions on matrix conditioning.
- **Poor scalability**: the method does not naturally extend to distributed or GPU-based architectures.
- **Limited hardware utilization**: modern CPUs rely heavily on vectorization and cache-aware block operations, which skyline-based scalar loops fail to exploit.
- **Fill-in inefficiency**: although constrained by the skyline profile, fill-in is not globally optimized through graph-based reordering techniques as in modern solvers.

In contrast, modern solvers such as MUMPS and Intel MKL PARDISO replace these limitations with advanced techniques including symbolic factorization, graph reordering, supernode or multifrontal decomposition, and heavy reliance on optimized dense linear algebra kernels.

3.4. Solver Benchmarking and Validation Sandbox

Before integrating external solvers directly into the original application, a dedicated benchmarking environment was developed in order to isolate and evaluate the performance of the different numerical solution strategies under controlled conditions.

The purpose of this environment was to:

- extract the stiffness matrix and right-hand side vector (`rhs`) from the original simulation;
- reconstruct the linear system externally;
- repeatedly solve the same problem using different solver implementations;
- measure execution times and numerical residuals consistently.

This approach provided a reproducible and simplified framework for comparing solver behavior without interference from the remaining components of the aeroelastic simulation.

3.4.1. Benchmark Configuration and Results

The benchmark was constructed using the stiffness matrix and right-hand side vector extracted from the original simulation at **time step 80**. Once exported, the same linear system was solved repeatedly for a total of **100 iterations** for each solver configuration.

For every test, the following quantities were measured:

- total execution time;
- average solution time per iteration;
- minimum and maximum execution times;
- residual ratio:

$$\frac{||A \cdot x - rhs||}{||rhs||}$$

The residual ratio was used as an indicator of numerical accuracy and solution stability. Table 3.3 summarizes the obtained results for the custom skyline solver, Intel MKL PARDISO, and MUMPS under different pivoting and factorization configurations.

Table 3.3: Comparison of solver performance on the benchmark linear system

Solver	**Average Time per Solve [ms]**	**Total Time [s]**	**Residual Ratio**
Custom Skyline LU Solver	48.16	4.815	9.65×10^{-14}
MKL PARDISO (standard pivoting)	23.97	2.397	6.72×10^{-14}
MKL PARDISO (aggressive pivoting)	22.86	2.286	1.50×10^{-10}
MUMPS (standard configuration)	17.63	1.763	9.70×10^{-14}
MUMPS (aggressive pivoting / symmetric configuration)	16.92	1.692	9.70×10^{-14}

The benchmark clearly shows that both external sparse direct solvers significantly outperform the original custom skyline LU implementation.

The custom solver required approximately:

$$32.93 \text{ ms} + 15.23 \text{ ms} \approx 48.16 \text{ ms}$$

per complete factorization and solution cycle, resulting in the highest execution time among all tested approaches.

Intel MKL PARDISO achieved a substantial reduction in execution time, approximately halving the computational cost compared to the custom implementation. This improvement is mainly attributable to:

- optimized sparse matrix storage (CSR);
- supernode-based factorization;
- efficient cache utilization;
- optimized BLAS Level 3 kernels.

Interestingly, enabling more aggressive pivoting and ordering optimizations slightly improved execution time further, although at the cost of a moderately larger residual ratio. Nevertheless, the obtained residual remained sufficiently small for engineering applications.

Among all tested configurations, MUMPS produced the best overall performance. The multifrontal strategy, combined with symbolic analysis and advanced ordering techniques, resulted in the lowest average execution time while maintaining excellent numerical accuracy.

A particularly interesting observation is that the aggressive pivoting configuration for MUMPS slightly improved performance without affecting the residual ratio significantly.

This suggests that, for the considered matrix structure, the symbolic analysis and multifrontal decomposition are highly effective at controlling fill-in and preserving numerical stability.

3.5. Integration of External Solvers into the Original Application

The benchmark results discussed in the previous section clearly demonstrated that modern sparse direct solvers are capable of providing significant reductions in computation time while maintaining numerical residuals comparable to those obtained with the original skyline LU implementation.

Following these observations, the next step of the work consisted in integrating the selected external solvers directly into the original aeroelastic simulation code.

3.5.1. Integration Challenges

Although the standalone sandbox environment allowed controlled testing and rapid validation, integrating the solvers within the complete production application proved substantially more complex.

From a software engineering perspective, this phase represented one of the most time-consuming activities of the entire project, second only to the migration of the original codebase toward x64 and UNIX-compatible environments.

Several challenges emerged during the integration process:

- adaptation of the original skyline matrix structures to the sparse formats required by the external solvers;
- management of memory ownership and allocation consistency between legacy code and external libraries;
- compatibility issues between old build systems and modern HPC numerical libraries;
- synchronization between solver calls and the internal time integration workflow;
- validation of numerical consistency with respect to the original implementation.

A significant amount of work was also required to correctly integrate the solver execution policies within the nonlinear solution procedure of the original code. In particular, the application performs multiple equilibrium iterations within the same time step in order to reach convergence. Recomputing a complete numerical factorization at every iteration would have introduced unnecessary overhead and significantly reduced the potential performance gains.

For this reason, additional logic was implemented to:

- perform new numerical factorizations only when explicitly required by the simulation workflow;

- reuse previously computed factorizations during convergence iterations within the same time step;
- separate the analysis, symbolic factorization, and numerical factorization phases according to the capabilities of the external solvers.

Since the sparsity pattern and structural connectivity of the stiffness matrix remain unchanged throughout the simulation, the **analysis phase** and the associated **symbolic factorization** were executed only once during the first iteration of the simulation. Subsequent time steps reused the same symbolic information, recomputing only the numerical factorization when matrix coefficients changed.

This strategy proved particularly important for both MUMPS and PARDISO, as symbolic analysis and ordering operations can represent a non-negligible computational cost if repeatedly executed unnecessarily.

Particular difficulties were encountered with **MUMPS**, whose installation and compilation process is strongly dependent on UNIX/Linux HPC environments, MPI libraries, BLAS/LAPACK implementations, and external ordering packages such as METIS or SCOTCH.

As a consequence, the solver could not be integrated directly into the original build chain. Instead, a separate compilation workflow was required, followed by dynamic linking with the rest of the project during a later stage of the build process.

3.5.2. Importance of the x64 Migration

The integration phase further confirmed that the migration from legacy x86 environments to modern x64 architectures was not simply an optional optimization step, but rather a strict requirement for modernization.

Both Intel MKL PARDISO and MUMPS are designed primarily for modern 64-bit systems and do not provide practical backward compatibility with older x86 execution environments. Consequently:

- the original x86 compilation model was incompatible with the external numerical libraries;
- memory limitations of 32-bit architectures would have become increasingly problematic for larger simulations;
- modern compiler optimizations and threading support are primarily targeted toward x64 architectures.

The successful migration to x64 therefore represented a necessary enabling step for all subsequent modernization activities. Once the integration issues were resolved, the complete application was executed using the new solver backends.

The results obtained within the fully integrated simulation environment were not perfectly identical to those observed in the standalone sandbox benchmark. This behavior was

expected, since the benchmark environment isolated only the linear system solution phase, while the complete simulation includes:

- dynamic matrix assembly;
- memory transfers and data conversions;
- time integration procedures;
- nonlinear iterations;
- additional synchronization and I/O overhead.

Nevertheless, the integrated application still exhibited performance improvements that remained broadly consistent with the trends observed during isolated benchmarking.

A detailed quantitative analysis of the final performance results, numerical behavior, and scalability characteristics will be presented later in Chapter *Results*.

3.6. Optimization of the Stiffness Matrix Assembly

The optimization activity on the structural solver was largely focused on the assembly of the global stiffness matrix and of the global right-hand-side vector. The objective was not to modify the finite element formulation, the nonlinear solution strategy, or the element-level physics, but rather to reduce the computational overhead associated with repeatedly transferring elemental contributions into the global algebraic structures.

The numerical model therefore remained unchanged; only the efficiency of the assembly procedure was improved.

3.6.1. Original Assembly Strategy

In the original implementation, the assembly phase followed a conventional element-by-element approach. At every nonlinear iteration:

- the global arrays were cleared;
- all structural elements were processed sequentially;
- elemental stiffness matrices and force vectors were recomputed;
- the correspondence between local elemental coefficients and global matrix positions was reconstructed dynamically during each assembly call.

Conceptually, the workflow can be summarized as follows:

```
ComputeStructuralMatrices
  -> ClearGlobalArrays

  -> for each element:
       BuildConnectivity(...)
       EvaluateElementKernel(...)
```

```
AssembleElement(...)
  -> recover destination indices
  -> verify active DOFs
  -> update global matrix and rhs
```

Although the elemental computations themselves are unavoidable in a nonlinear finite element problem, profiling showed that a large portion of the assembly cost originated from purely administrative operations:

- repeated reconstruction of destination indices;
- repeated retrieval of matrix metadata;
- branch-heavy inner loops;
- repeated processing of connectivity information.

The same topological information was therefore rediscovered thousands of times during the simulation despite remaining invariant throughout the analysis.

3.6.2. Progressive Optimization Strategy

The optimization strategy was based on several observations regarding the structure of the problem:

1. the elemental connectivity pattern does not change during the simulation;
2. the sparsity structure of the global stiffness matrix remains fixed after initialization;
3. only the numerical values of the local matrices vary with time, not their destinations in the global matrix;
4. most of the overhead inside the assembly kernel is related to index management and control logic rather than numerical computation.

These considerations suggested that the assembly process could be transformed from a repeated destination-recovery procedure into a direct scatter operation based on precomputed mappings.

The optimization was introduced progressively through several stages.

Reduction of avoidable hot-path operations: The first improvements consisted in removing unnecessary operations from the most frequently executed loops:

- auxiliary data retrieval was moved outside the hot path whenever possible;
- conditional branches inside deeply nested loops were minimized;
- repeated access to matrix structural information was reduced.

Although individually small, these modifications reduced the overhead of the assembly kernel without altering the numerical formulation.

Introduction of reusable scatter mappings: The second step introduced reusable scatter maps. Instead of rebuilding the relation between local elemental entries and global matrix positions during every assembly call, the destination indices were computed only once and then reused throughout the simulation.

Initially, this mechanism relied on connectivity-based lookup tables. However, further analysis showed that the structural entity itself represented a stable identifier of the connectivity pattern.

The final implementation therefore attached the scatter information directly to the structural entities, transforming destination retrieval into a constant-time operation.

Packed scatter representation: The most important optimization consisted in replacing the original dense scatter loops with a packed representation containing only effective contributions.

In the original implementation:

- the assembly loop visited all local matrix entries;
- invalid or inactive contributions were discarded through branches.

In the optimized implementation:

- only valid contributions are stored;
- branch checks disappear from the hot loop;
- the number of iterations is reduced from the full local matrix size to the number of actual non-zero contributions.

The assembly process therefore becomes essentially a direct accumulation of local values into already known global destinations.

Optimized Stiffness Matrix Assembly Workflow
1. Global Initialization
- Reset of SparseMatrixValues and Skyline storage
- Reset of RHS vector
2. Element Loop
- Iterate over all element families and finite elements
- Compute element connectivity (only if required)
- Evaluate element kernel $\rightarrow$ local stiffness matrix + local force vector
3. Entity-Based Scatter Lookup (Optimization Core)
- Retrieve scatter structure using **hash map lookup**
- Access precomputed entity-level metadata in $O(1)$ time
- No reconstruction of connectivity-to-global mapping
4. Packed Fast Scatter Operation
- Direct mapping local $\rightarrow$ global indices
- Update only valid contributions (packed representation)
- Conditional stiffness update only when required by factorization step
- Branch-free inner scatter loop
5. Global Assembly Update
- Accumulation of local contributions into global sparse structures
- Skyline structure no longer used in hot path
Key Optimization Outcome
- Elimination of runtime index reconstruction
- Replacement of connectivity search with hash map entity lookup
- Reduction of assembly to direct memory updates
- Significant reduction of administrative overhead

Figure 3.1: Workflow of the optimized stiffness matrix assembly based on entity-level hash map lookup and packed scatter representation.

Aspect	Original path	Optimized path
Destination recovery	Reconstructed during each assembly call	Built once and reused as persistent scatter metadata
Lookup key	Implicitly derived from current connectivity at runtime	Directly associated with the structural entity and local call identifier
Hot-path data access	Includes repeated extraction of fallback data	Limited to data actually needed in the active path
Inner stiffness loop	Dense loop over all local entries	Packed loop over contributing entries only
Branching	Branch evaluated inside the local scatter loop	Branch removed from the packed hot path
Global destination access	Recovered through repeated index processing	Reused through precomputed destinations
Per-call overhead	Significant administrative work in addition to updates	Mostly reduced to accumulation of local contributions
Observed effect	Assembly remained a visible hotspot	Assembly no longer dominates the runtime profile

Table 3.4: Compact comparison between the original and optimized assembly paths.

3.6.3. Optimization of Sparse Format Generation

An additional bottleneck emerged after the integration of the external sparse direct solvers.

The original application stores the global stiffness matrix using a skyline representation, while the new solvers require:

- CSR format for Intel MKL PARDISO;
- coordinate (COO) format for MUMPS.

In the first implementation, the conversion process followed the sequence:

$$\texttt{Skyline} \rightarrow \texttt{Full Matrix} \rightarrow \texttt{CSR / COO}$$

Moreover, this conversion was executed repeatedly during the simulation iterations.

Although correct from a functional perspective, this approach introduced a very large overhead:

- reconstruction of a temporary dense matrix generated unnecessary memory traffic;
- sparse structure reconstruction was repeated continuously;
- conversion costs partially masked the gains introduced by the external solvers.

To solve this issue, dedicated initialization routines were developed and executed only once during the first simulation step.

These routines generate a set of mapping structures capable of directly associating every skyline matrix coefficient with its corresponding position inside the destination CSR or coordinate arrays.

After the initialization phase:

- the skyline structure is no longer updated;
- CSR and COO matrices become the primary storage structures;
- coefficient updates are performed directly on the target sparse arrays through the precomputed mappings.

This introduces only a very small initialization overhead during the first iteration, while completely removing the repeated skyline-to-sparse conversion cost from the simulation loop.

As a consequence, the performance improvements introduced by MUMPS and Intel MKL PARDISO become much more evident within the fully integrated application.

3.6.4. Final Results

The final implementation confirmed the initial optimization hypotheses.

The updated assembly strategy achieved several important improvements:

- removal of repeated destination reconstruction;
- elimination of unnecessary branches inside hot loops;
- reduction of administrative overhead during assembly;
- elimination of repeated skyline-to-sparse conversion costs;
- improved memory locality and sparse structure reuse.

Most importantly, none of these optimizations altered the mathematical formulation of the problem. The same elemental matrices, global algebraic system, and nonlinear solution procedure are preserved. Only the efficiency of the transfer mechanisms and sparse storage management was improved.

From a methodological perspective, this optimization activity highlights the importance of exploiting invariants already present in the numerical model. By transforming assembly and sparse-format generation into reusable metadata-driven operations, the computational overhead associated with matrix construction was substantially reduced.

Finally, profiling results indicate that the main computational bottlenecks have progressively shifted away from matrix assembly toward the actual aeroelastic and element-level physics computations. This represents an important result, since it suggests that future

optimization efforts should increasingly target the physical kernels of the simulation rather than the algebraic infrastructure.

4 | Results

This chapter presents the results obtained from the various optimization and modernization activities discussed throughout this work. In order to ensure a fair comparison between different implementations, solvers, and optimization strategies, all simulations were performed using the same wind turbine model and the same operating conditions. Since the objective of this study is the evaluation of computational performance rather than the analysis of the turbine itself, the specific geometric and structural characteristics of the model are not discussed in detail.

All tests were conducted using the **DLC 1.1** operating condition. DLCs (*Design Load Cases*) are standardized simulation scenarios defined by international wind turbine certification standards, such as IEC 61400 and GL guidelines. To achieve certification, a wind turbine must be analyzed under a wide range of operating and environmental conditions, including normal power production, turbulent and gusty winds, fault situations, emergency shutdowns, parked conditions, and commissioning or decommissioning phases. As a result, a complete certification campaign typically requires hundreds or even thousands of dynamic simulations.

Among these scenarios, **DLC 1.1** is generally considered the most representative of the normal operating behavior of a wind turbine. It corresponds to *normal power production under the Normal Turbulence Model (NTM)*, with hub wind speeds ranging between cut-in and cut-out conditions. Because of its representative nature and its relevance for both fatigue and ultimate load assessments, DLC 1.1 is commonly used as a reference case in aeroelastic analyses.

4.1. Validation

Before discussing the performance improvements achieved through the proposed optimizations, it is essential to verify that the modified application produces results consistent with those generated by the original implementation. In the context of scientific computing, performance gains have little practical value if they are obtained at the expense of numerical accuracy or by altering the physical behavior predicted by the simulation.

For this reason, the first step of the analysis consists of a validation campaign aimed at assessing the numerical consistency of the modernized code. The objective is not to obtain bitwise-identical results, since differences in hardware architectures, compiler optimizations, floating-point arithmetic, and solver implementations naturally introduce small numerical variations. Instead, the goal is to verify that all tested configurations

produce equivalent physical responses and that any observed discrepancies remain within acceptable engineering tolerances.

Particular attention is devoted to the integration of the external sparse direct solvers. Although both MUMPS and Intel MKL PARDISO solve the same algebraic problem as the original skyline-based LU solver, they employ different matrix storage formats, ordering strategies, pivoting techniques, and factorization algorithms. Consequently, small differences in the numerical solution are expected and must be carefully evaluated before proceeding with any performance analysis.

The following sections compare the results obtained using the three solver configurations considered throughout this work:

- **Solver 0**: original custom skyline LU solver;
- **Solver 2**: MUMPS sparse direct solver;
- **Solver 3**: Intel MKL PARDISO sparse direct solver.
- **LB**: version with aggressive pivoting optimization

Unless otherwise specified, all plots and numerical comparisons presented in this validation chapter adopt this solver numbering convention.

4.1.1. Residuals

A first validation metric for the numerical solvers is based on the analysis of the residual norm associated with the linear system solution. At each time step, the accuracy of the computed solution vector is evaluated through the following relative residual measure:

$$r = \frac{\|A\,x - b\|_2}{\|b\|_2}$$

where A is the global stiffness matrix, x is the computed displacement (or state) vector, and b is the right-hand side vector.

This quantity provides a direct indication of how accurately each solver satisfies the underlying linear system. Lower values correspond to higher numerical precision.

The comparison is performed over the first 500 time steps of the simulation, considering the solution at the first nonlinear iteration of each step in order to ensure consistency of the analysis across all solvers.

As shown in Figure 4.1, all solvers consistently maintain residual values below 10^{-9}, confirming the numerical stability of the implementations.

More specifically:

- The **custom skyline LU solver (Solver 0)** provides a reference accuracy of approximately 6×10^{-13}, remaining extremely stable throughout the simulation.

Figure 4.1: Comparison of residual norms over the first 500 time steps for the different solvers.

- **MUMPS (Solver 2)** achieves comparable accuracy, remaining on the order of 10^{-13}, with negligible sensitivity to pivoting options.
- **Intel MKL PARDISO (Solver 3)** yields slightly higher residuals around 7–8 $\times$ 10^{-13} in its default configuration, while the introduction of more aggressive pivoting strategies improves performance at the cost of reduced accuracy, shifting the residual level to approximately 10^{-10}. Despite this degradation, the results remain well within acceptable engineering tolerances.

Solver	Residual magnitude	Remarks
0 (Custom LU)	$\sim 6 \times 10^{-13}$	Reference solution, highest accuracy
2 (MUMPS)	$\sim 10^{-13}$	Stable, insensitive to pivoting options
3 (PARDISO - default)	$\sim$ 7–8 $\times$ 10^{-13}	Comparable to reference
3 (PARDISO - aggressive pivoting)	$\sim 10^{-10}$	Slight accuracy degradation, still acceptable

Table 4.1: Summary of residual levels for the different solvers.

Overall, the residual analysis confirms that all tested solvers produce numerically consistent solutions, thereby validating their use for the subsequent performance evaluation.

4.1.2. Displacements

While the residual analysis provides information about the accuracy of the linear system solution, the ultimate validation criterion for the present application is the comparison of the physical quantities produced by the simulation.

At each nonlinear iteration, the structural problem is formulated as

$$K\,\Delta x = f_{eq}$$

where the solution vector Δx represents the displacement increment associated with the current time step. The actual structural configuration is then obtained by accumulating these increments throughout the simulation:

$$x(t_n) = x(t_{n-1}) + \Delta x(t_n)$$

As a consequence, even small numerical differences introduced by different linear solvers may accumulate over time and potentially alter the global structural response. For this reason, the displacement histories generated by the various solver configurations were compared over the entire simulation.

Several tens of nodes distributed throughout the structure were analyzed. For each degree of freedom, the discrepancy with respect to the reference solution (Solver 0) was evaluated through the following absolute error measures:

$$err_{0-2} = |x_0 - x_2|$$

$$err_{0-3} = |x_0 - x_3|$$

where:

- x_0 is the displacement obtained using the original custom skyline solver;
- x_2 is the displacement obtained using MUMPS;
- x_3 is the displacement obtained using Intel MKL PARDISO.

For the vast majority of the analyzed degrees of freedom, the observed errors remain two to three orders of magnitude smaller than the corresponding displacement values. From an engineering perspective, these differences are negligible and confirm that the physical behavior of the system is preserved.

For displacement components exhibiting predominantly monotonic trends, the agreement is even stronger, with errors becoming almost indistinguishable from numerical noise. However, some highly periodic components require a more detailed analysis, as they exhibit error patterns that are not immediately explained by the residual analysis alone.

To illustrate this behavior, two representative nodes were selected among the approximately 720 nodes composing the model. The first example corresponds to Node 250.

Node 250 Displacement Histories and Error Analysis

Figure 4.2 reports the six displacement and rotation components of Node 250.

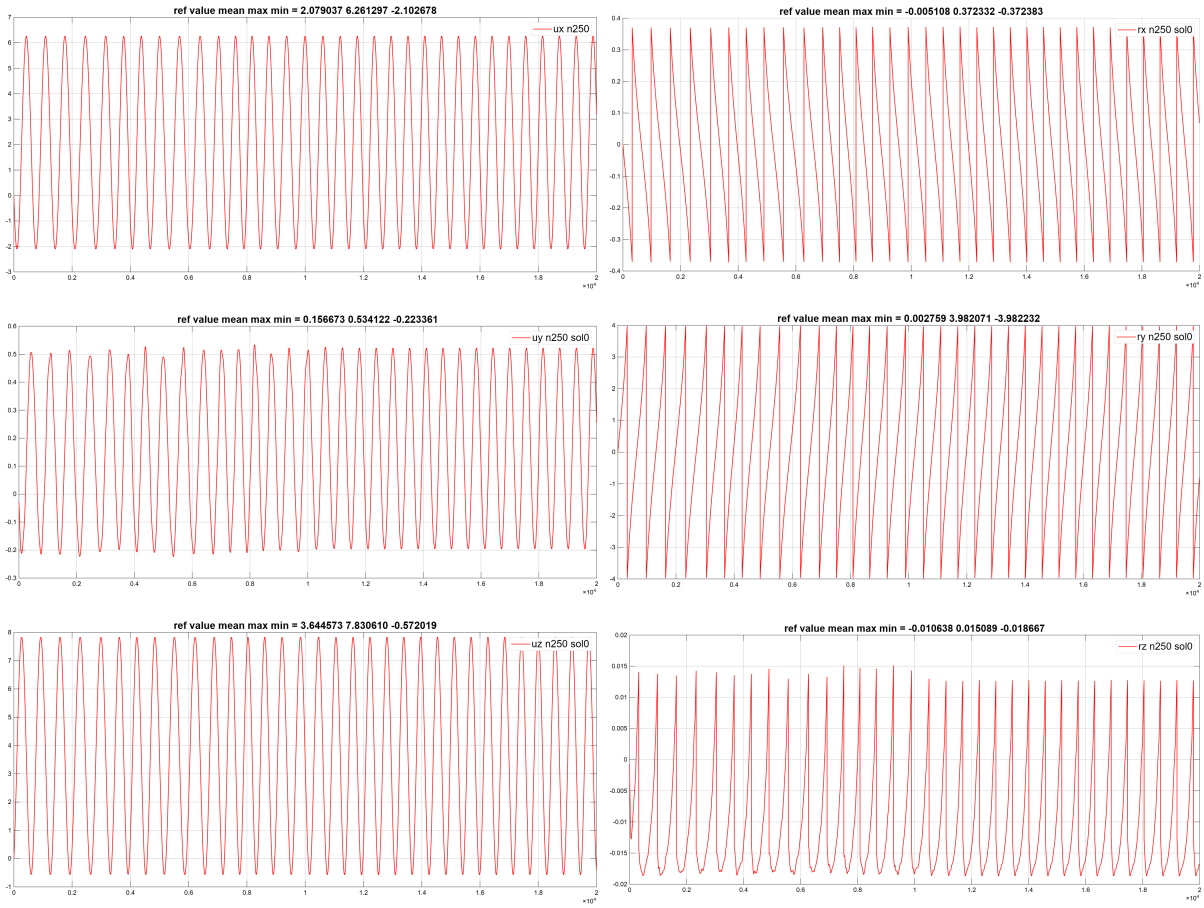

Figure 4.2: Displacement and rotation histories of Node 250.

Visual inspection shows that the solutions produced by the custom solver are periodic with almost fixed period and amplitude.

The corresponding absolute errors are reported in Figure 4.3.

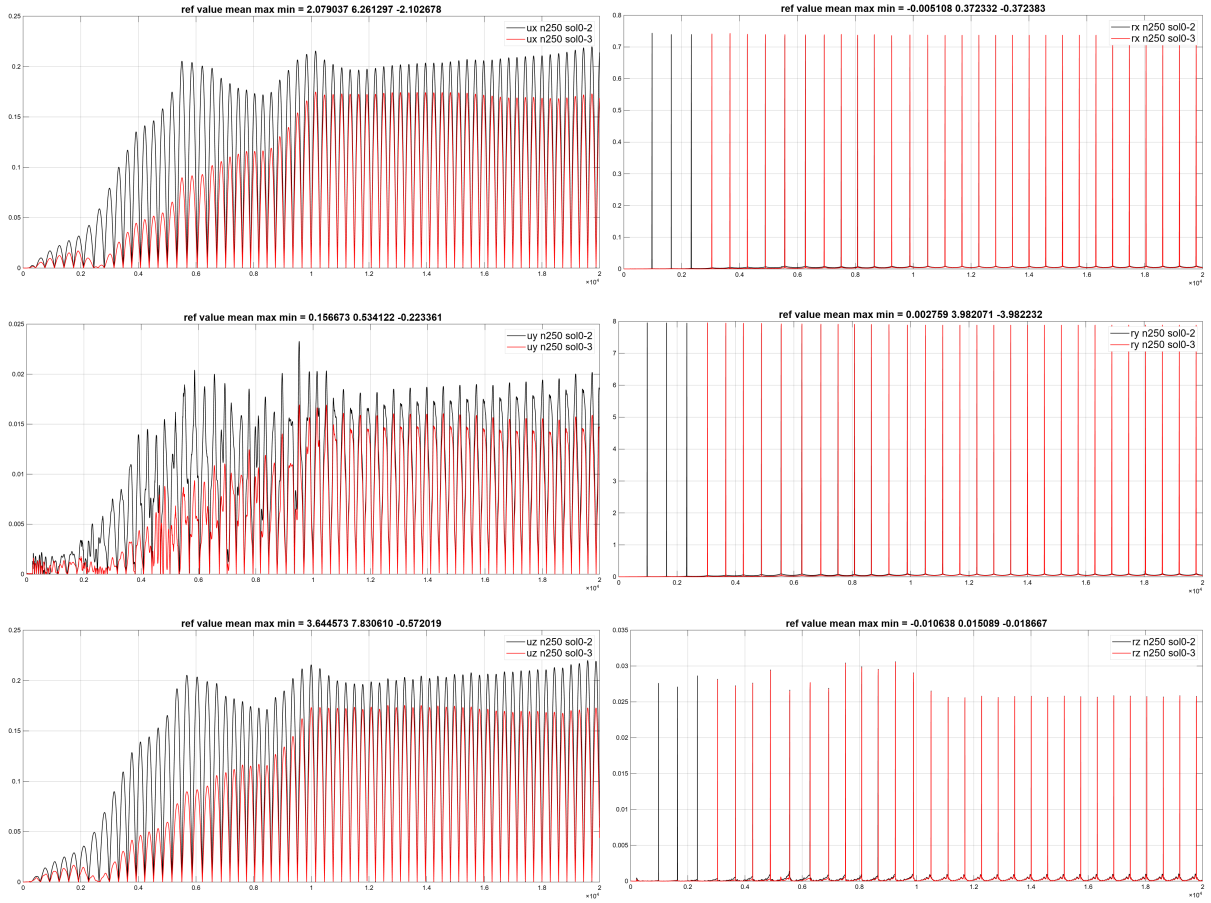

Figure 4.3: Absolute errors with respect to the reference solution for Node 250.

At first glance, the rotational degrees of freedom appear to exhibit errors of the same order of magnitude as the corresponding rotations. Similarly, the translational components show periodic error patterns that remain visible despite being smaller than the actual displacement amplitudes.

This behavior might initially suggest the presence of significant numerical discrepancies. However, a more detailed inspection reveals a different interpretation. To better understand the origin of these errors, Figure 4.4 presents three magnified views of the u_x displacement component.

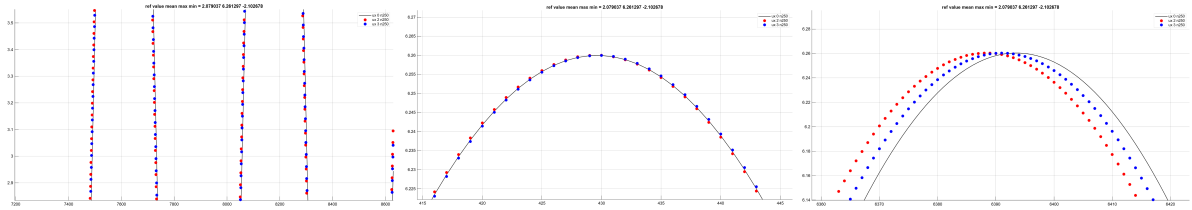

Figure 4.4: Magnified views of the u_x displacement component for Node 250.

The enlarged views clearly show that the solutions produced by MUMPS and PARDISO

follow the same physical evolution as the reference solver. In particular, Figure 4.4 demonstrates that all solvers reproduce the same oscillatory behavior, the same amplitudes, and the same overall dynamics.

The apparent errors originate from a very small phase shift that progressively develops during the simulation. While the solutions are almost perfectly coincident during the initial time steps, as illustrated by the central detail, after approximately 15 000 time steps the solutions generated by the external solvers reach the same physical state one or two time steps earlier than the reference implementation.

As a consequence:

- errors become almost zero near local maxima and minima, where the displacement derivative is small;
- larger errors appear in regions with steep slopes, where even a tiny phase difference produces a visible pointwise discrepancy;
- the physical response remains essentially unchanged.

Therefore, the observed differences should not be interpreted as a loss of numerical accuracy. Rather, they correspond to a slight temporal phase shift between solutions that otherwise exhibit the same structural behavior. For the engineering objectives of the simulation, such differences are negligible and do not affect the validity of the computed response.

Node 650 Displacement Analysis

As a complementary example, the response of Node 650 is also considered. Unlike Node 250, whose behavior is dominated by a highly periodic response, Node 650 exhibits a more irregular and less repetitive displacement history, although the overall displacement amplitudes remain relatively small.

For this node, only the u_x displacement component is analyzed, since the remaining translational and rotational degrees of freedom are characterized by very small oscillations around zero and do not provide additional information regarding the comparison between solvers.

Figure 4.5 reports both the displacement history and the corresponding error with respect to the reference solution.

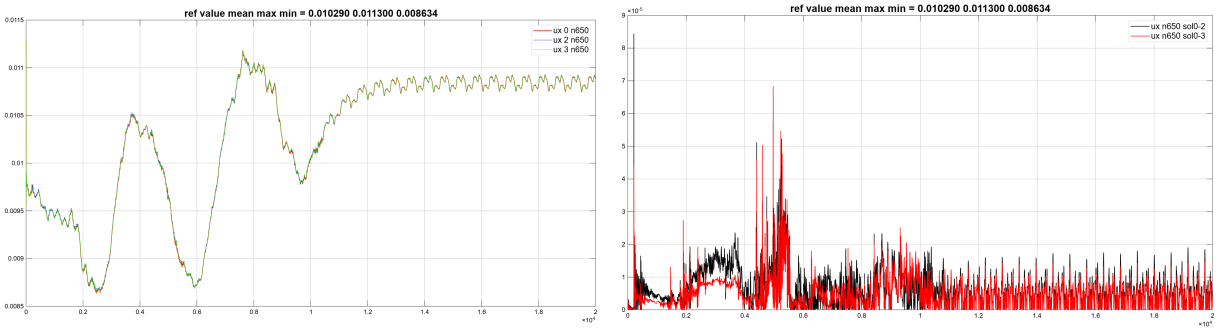

Figure 4.5: Node 650: u_x displacement history (left) and corresponding absolute error (right).

The results confirm the observations already made for Node 250. Despite the relatively small displacement amplitudes involved, the solutions produced by MUMPS and MKL PARDISO remain almost perfectly superimposed on the reference solution throughout the simulation.

The error magnitude remains consistently small, typically two orders of magnitude lower than the displacement itself. From an engineering standpoint, such differences are negligible and indicate that the modernization of the linear solver does not alter the global structural response.

To further investigate the behavior of the solutions, a magnified view of a representative portion of the displacement history is reported in Figure 4.6.

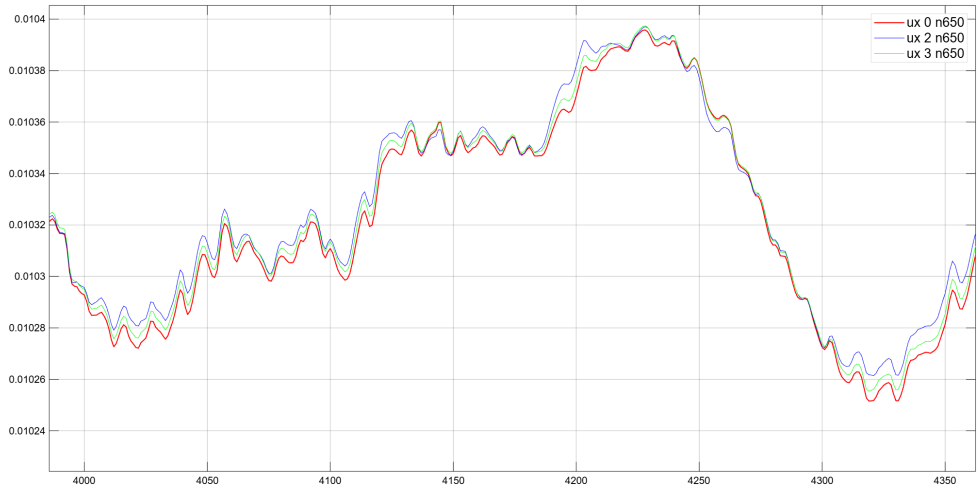

Figure 4.6: Magnified view of the u_x displacement component for Node 650.

The enlarged view clearly shows that both external solvers accurately reproduce the physical evolution of the displacement. Even for very small displacement values, the solutions follow the same trends, local oscillations, and transient behavior observed in the original implementation.

A slight tendency to overestimate the displacement can be observed for both MUMPS and MKL PARDISO. However, the magnitude of this deviation remains extremely limited and does not significantly affect the overall response of the system. The three solutions therefore remain fully consistent from an engineering perspective.

4.1.3. Control Variables Time Histories

A final validation step was performed by comparing the time histories of several control and operational variables generated during a complete aeroelastic simulation.

A 1000-second DLC 1.1 simulation was executed using both the original custom solver and the MKL PARDISO implementation. The resulting output files were then processed using the standard post-processing tools provided with CP-Lambda. This approach allows the comparison to be extended beyond purely numerical quantities such as residuals and nodal displacements, evaluating instead variables that are directly related to the turbine control system and overall machine behavior.

The results are particularly important because these quantities are often used for design verification, controller tuning, fatigue assessment, and certification activities. Consequently, even small deviations could potentially reveal differences in the global dynamic response of the model.

Overall, the comparison shows excellent agreement between the two implementations, with discrepancies remaining extremely limited throughout the entire simulation.

Electrical Power and Electrical Torque: Figure 4.10 compares the electrical power and electrical torque time histories together with magnified views of representative sections of the signals.

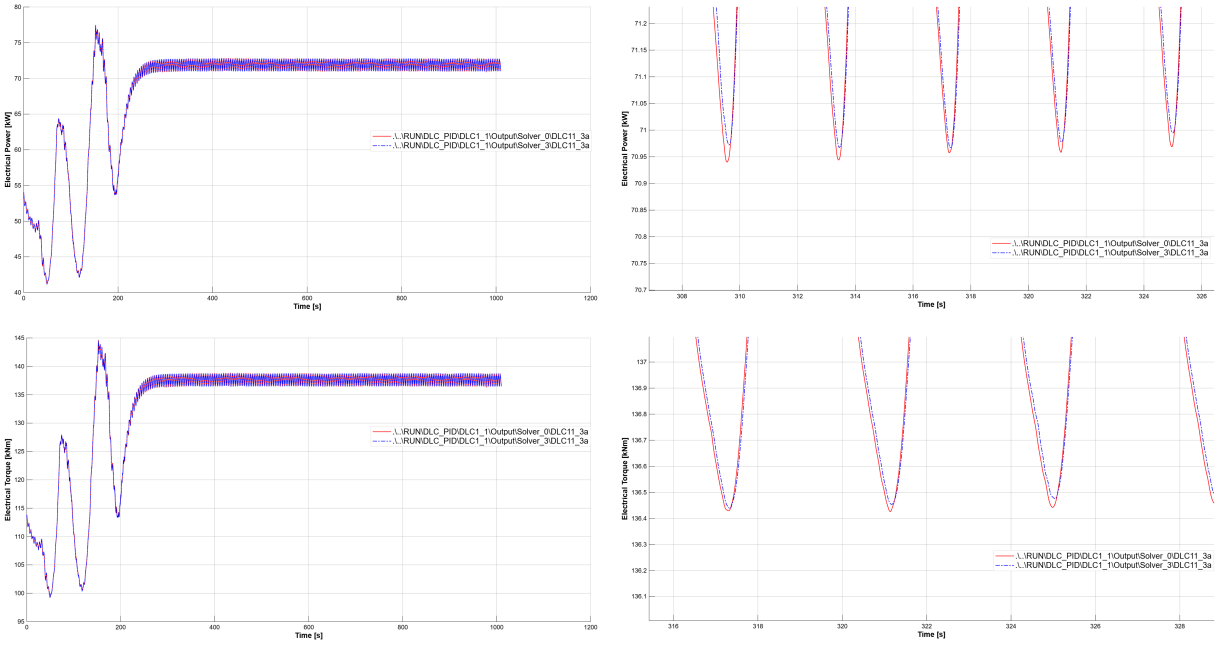

Figure 4.7: Electrical power and electrical torque comparison between the original solver and MKL PARDISO.

The global trends are visually indistinguishable over the entire simulation duration. The magnified views reveal only very small differences between the two solutions.

The estimated relative error remains approximately:

- 0.08% for electrical power;
- 0.06% for electrical torque.

Such values are significantly below the uncertainty typically associated with engineering simulations of complex aeroelastic systems and can therefore be considered negligible.

Demanded Generator Torque and Yaw Error: A second comparison was performed on the demanded generator torque and the yaw error, shown in Figure 4.8.

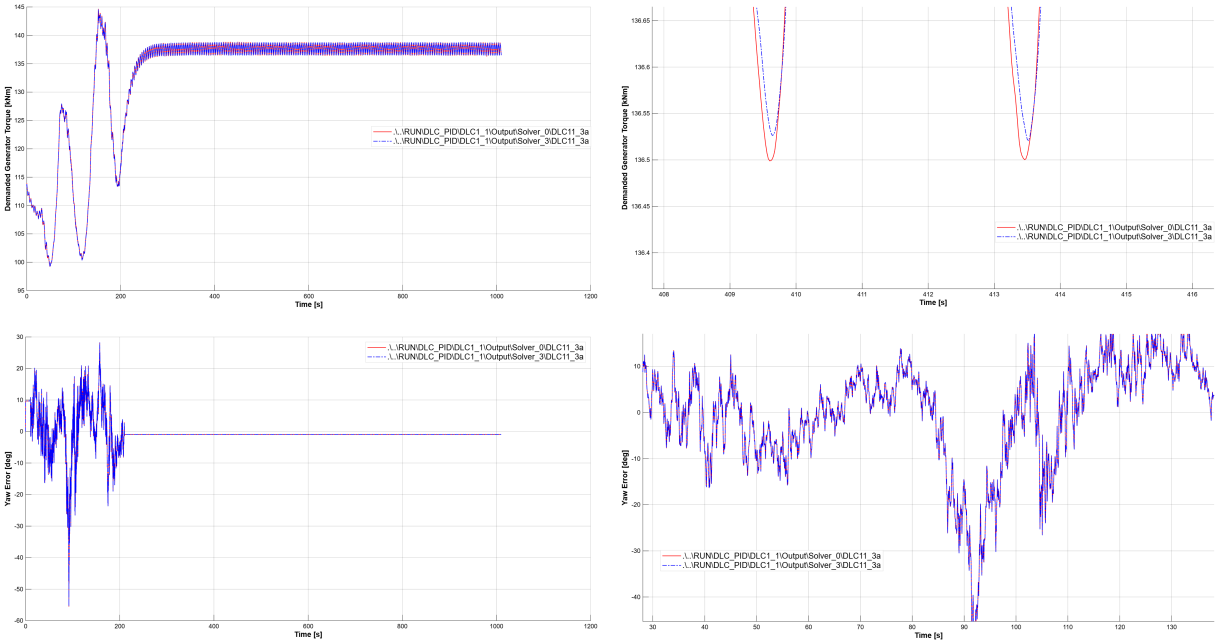

Figure 4.8: Demanded generator torque and yaw error comparison.

The demanded generator torque exhibits an even smaller discrepancy, with an estimated relative error of approximately 0.01%.

The yaw error signal, on the other hand, appears completely superimposed for the entire simulation. No meaningful differences can be identified either in the full signal or in the enlarged view, indicating that the controller behavior is preserved after the solver replacement.

Rotor Speed and Filtered Rotor Speed: Among the available control variables, rotor speed represents one of the most significant validation indicators. This quantity results from the combined effect of aerodynamic loads, structural dynamics, controller actions, drivetrain dynamics, and generator response. Consequently, any substantial numerical inconsistency introduced by the solver would be expected to appear in this signal.

Figure 4.9 compares the rotor speed and filtered rotor speed histories.

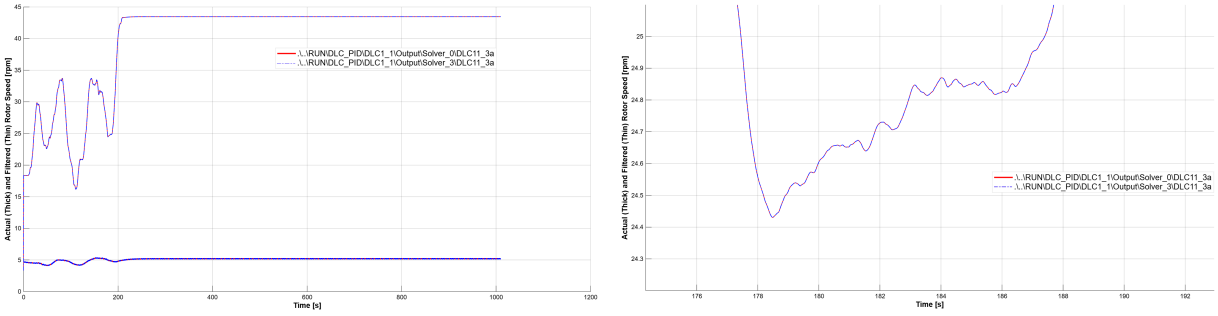

Figure 4.9: Rotor speed and filtered rotor speed comparison.

The two solutions are virtually identical. Both the global behavior and the local fluctuations are reproduced with no observable differences. Given the physical significance of this variable, this result provides strong evidence that the modifications introduced in the structural solver do not alter the global aeroelastic response of the wind turbine model.

Table 4.2 summarizes the observed discrepancies for the analyzed control variables.

Variable	Estimated Error	Agreement	Remarks
Electrical Power	$\approx 0.08\%$	Excellent	Smallest deviations visible only in magnified views
Electrical Torque	$\approx 0.06\%$	Excellent	Practically identical dynamic response
Demanded Generator Torque	$\approx 0.01\%$	Excellent	Differences close to numerical noise
Yaw Error	$\approx 0\%$	Perfect	Signals fully superimposed
Rotor Speed	$\approx 0\%$	Perfect	Identical global and local behavior
Filtered Rotor Speed	$\approx 0\%$	Perfect	No observable discrepancy

Table 4.2: Summary of control variable comparison between the original solver and MKL PARDISO.

The results obtained from the post-processing analysis are fully consistent with the conclusions drawn from the residual and displacement comparisons. The adoption of modern sparse direct solvers introduces only negligible differences in the computed quantities while preserving the physical behavior of the aeroelastic model.

From an engineering perspective, the modified implementation can therefore be considered fully reliable. The observed discrepancies remain well below thresholds that could affect design decisions, load assessments, controller performance evaluations, or certification-oriented analyses. Consequently, the modernization of the solver infrastructure appears to preserve the numerical fidelity of the original code while enabling the performance improvements discussed in the next sections

4.2. Computational Performance Environment

Before analyzing the performance results, it is necessary to define the hardware and software environment on which all benchmarks have been executed. The objective of this section is not to provide an exhaustive hardware characterization, but rather to highlight the key architectural features that directly influence the computational behavior of the numerical simulations, in particular execution time, memory bandwidth, and solver scalability.

The first set of experiments was carried out on a Windows-based workstation running **Microsoft Windows 11 Home (build 26200)** on a x64 architecture. The most relevant computational characteristics for the present analysis are summarized below:

- **CPU**: Intel Core i7-1360P (13th Gen)
- **Cores / Threads**: 12 physical cores, 16 logical processors
- **Base frequency**: ~ 2.2 GHz

- **RAM**: 16 GB DDR5 (6400 MHz)
- **Storage**: NVMe SSD (approx. 512 GB class)
- **Architecture**: x64-based system

This machine is used as the primary reference platform for evaluating and comparing the following execution environments:

- native execution on Windows 11;
- execution under Windows Subsystem for Linux (WSL);
- execution inside Docker containers running on Windows.

A comparison benchmark was also performed on a native Linux machine with the following specifications:

- **CPU**: Intel Core i3-1115G4 processor
- **Cores / Threads**: 2 physical cores, 4 logical threads
- **Base frequency**: 4.1 GHz
- **RAM**: 8 GB of RAM
- **Architecture**: Ubuntu 24.04.4 LTS operating system

In all cases, the same numerical model, solver configurations, and simulation scenarios are used.

4.3. Timing Performance Analysis

After validating the numerical consistency of the modified code, the next step consists in evaluating the impact of the introduced optimizations on the overall execution time.

The simulations presented in this section were executed under **Windows Subsystem for Linux (WSL)** using the hardware platform described previously. Four analyses of increasing duration were considered, corresponding to approximately 1000, 5000, 10000 and 20000 time steps. For each run, timing statistics were collected for:

- stiffness matrix assembly;
- matrix factorization;
- linear system solution;
- total sparse solver time (factorization + solution);
- total simulation elapsed time.

Since all simulations solve exactly the same physical problem, the absolute execution times are strongly dependent on the specific hardware and software environment and therefore have limited standalone significance. More meaningful information is obtained by comparing the relative differences between the solvers.

It is also worth noting that the matrix assembly phase was originally designed around the skyline storage format used by the legacy solver. Although the assembly kernel was significantly optimized as discussed in Chapter 3, an additional mapping step is still required when MUMPS and PARDISO are employed. The elemental contributions are assembled directly into CSR or Coordinate structures through precomputed index maps, but this process still introduces a small overhead compared with the original skyline-only workflow. Consequently, the assembly time remains largely independent of the selected solver and should not be considered when evaluating the efficiency of the linear algebra backend itself.

4.3.1. Average Solver Performance and Relative Speedup

Table 4.3 summarizes the average timings measured for the largest simulation (approximately 20000 time steps), which provides the most statistically representative dataset.

Table 4.3: Average timing statistics for the 20000-step simulation.

Metric	**Custom LU**	**MUMPS**	**PARDISO**
Factorization [ms]	3.80	5.77	1.88
Solution [ms]	3.79	0.48	0.26
Total Solver [ms]	5.73	3.37	1.21
Assembly [ms]	2.72	2.88	2.67
Total analysis time [s]	429.6	337.3	239.0

Several observations can immediately be drawn.

First, the custom skyline solver exhibits similar costs for factorization and solution. This behavior is expected because both phases rely on sequential skyline traversals and neither operation benefits from advanced sparse-matrix optimizations.

Second, MUMPS performs a more expensive factorization than the custom solver, but compensates with an extremely efficient triangular solve phase. As a result, the overall solver time decreases substantially.

Finally, PARDISO consistently delivers the best performance in both factorization and solution, resulting in the lowest overall solver cost.

To better quantify the benefit of the new implementations, Table 4.4 reports the percentage reduction of the total solver time relative to the original skyline solver.

Table 4.4: Relative improvement with respect to the custom skyline solver.

Solver	**Total Solver Time [ms]**	**Improvement**
Custom LU	5.73	—
MUMPS	3.37	41.2%
PARDISO	1.21	78.9%

The results clearly show that both external sparse solvers outperform the original skyline implementation.

MUMPS reduces the linear solver cost by approximately 40%, while PARDISO achieves a reduction close to 80%. The latter result is particularly significant considering that no modifications were made to the finite-element formulation itself; the gain derives almost entirely from the adoption of a modern sparse direct solver together with an efficient sparse storage format.

Table 4.5: Total simulation times and speedup relative to the custom LU solver on Linux-WSL.

Time Steps	Custom LU [s]	MUMPS [s]	MUMPS Gain	MKL [s]	MKL Gain
1000	18.84	17.30	8.2%	12.67	32.8%
5000	98.40	84.91	13.7%	63.73	35.2%
10000	205.80	172.73	16.1%	128.10	37.8%
20000	429.60	337.31	21.5%	238.99	44.4%

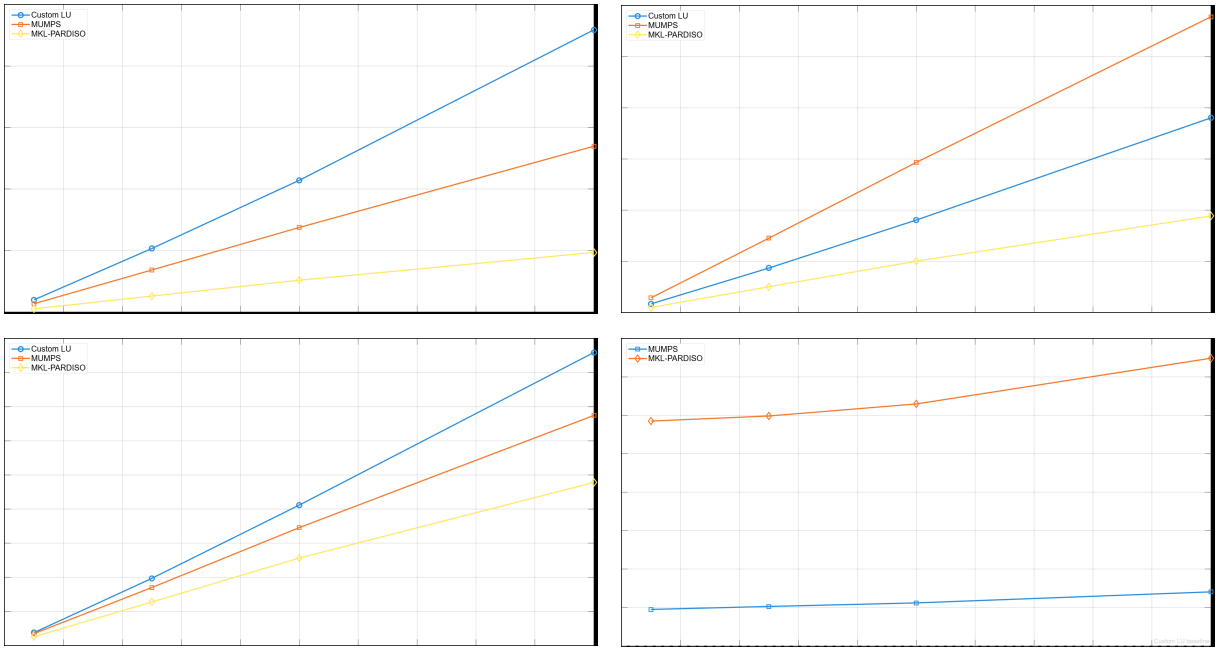

Figure 4.10: Scaling behaviour of solution, factorization, entire solver, and relative speed-up

An important aspect of the collected statistics is their consistency across all tested simulation lengths.

For the custom solver, the average factorization time increases from approximately 3.3 ms to 3.8 ms as the simulation progresses, while the average solver time grows from 4.9 ms to 5.7 ms. This trend suggests increasing pressure on memory hierarchy and cache efficiency as the execution length increases.

In contrast, both MUMPS and PARDISO maintain remarkably stable average timings over all simulations:

- MUMPS remains close to 3.4 ms per complete solve.
- PARDISO remains close to 1.25–1.30 ms per complete solve.

This stability indicates a much better exploitation of modern sparse linear algebra kernels and memory access patterns.

4.3.2. Assembly versus Solver Cost

One particularly interesting consequence of the introduced optimizations is the shift of the computational bottleneck.

Before the modernization effort, the skyline solver dominated the execution profile. After the introduction of CSR/Coordinate storage and modern sparse solvers, the linear algebra cost was reduced to the point that the assembly phase became comparable to, or even more expensive than, the solve phase itself.

For the PARDISO implementation, the average assembly time is approximately:

$$t_{\text{assembly}} \approx 2.67 \text{ ms}$$

while the complete solver requires only:

$$t_{\text{solver}} \approx 1.21 \text{ ms}.$$

In other words, the stiffness matrix assembly now costs more than twice the time required to factorize and solve the linear system.

This observation confirms two important conclusions:

1. the skyline solver was indeed the primary bottleneck of the original implementation;
2. after introducing modern sparse solvers, further performance gains are more likely to come from assembly and physics-level optimizations than from additional work on the linear algebra backend.

Overall, the results demonstrate that the migration from the legacy skyline LU solver toward modern sparse direct solvers produces substantial computational benefits while preserving the numerical accuracy demonstrated in the validation phase. Among the tested alternatives, PARDISO provides the most favorable compromise between robustness and computational efficiency, reducing the linear solver cost by nearly a factor of five with respect to the original implementation.

4.3.3. Docker Container Evaluation

In order to assess the portability of the proposed optimizations and to investigate the impact of containerization on the overall execution time, an additional benchmark campaign was performed using a Docker container based on Ubuntu 22.04.5. For simplicity, only the original custom LU solver and the MKL-PARDISO implementation were considered in this comparison, since the previous analyses had already shown that PARDISO provides the best results.

Initial executions were performed using bind-mounted working directories located on the Windows host system. Although the execution times of the numerical kernels remained comparable to those observed under Linux-WSL, the overall simulation time increased dramatically due to the cost of file-system synchronization between the container and the host operating system. In some cases, execution times were approximately three times larger than expected. To eliminate this source of overhead, all simulation files were copied directly inside the container filesystem before execution. Table 4.6 summarizes the most relevant timing statistics.

Table 4.6: Average timings measured inside the Docker container.

Steps	Solver	Factorization [ms]	Solution [ms]	Solver Total [ms]	Simulation [s]
1000	Custom	3.35	3.05	4.75	19.3
	PARDISO	1.71	0.24	1.10	11.3
5000	Custom	3.29	2.97	4.64	93.2
	PARDISO	1.80	0.25	1.15	58.7
10000	Custom	3.39	3.11	4.83	193.2
	PARDISO	1.99	0.27	1.27	129.2
20000	Custom	3.80	3.37	5.30	418.3
	PARDISO	2.02	0.28	1.29	256.3

The scaling behavior remains essentially linear for both solvers, confirming that neither the containerization layer nor the adopted matrix conversion strategy introduce significant nonlinear overheads as the simulation length increases. Using the 20000-step simulation as the largest and most representative workload, the relative improvements of MKL-PARDISO with respect to the custom skyline solver are summarized in Table 4.7.

Table 4.7: Performance improvement of MKL-PARDISO over the custom solver (20000 steps).

Metric	Custom LU	Improvement
Factorization time	3.80 ms	46.8%
Solution time	3.37 ms	91.7%
Total solver time	5.30 ms	75.7%
Total simulation time	418.3 s	38.7%

The results are consistent with those obtained under Linux-WSL. Is also provided a direct comparison with the Linux-WSL results in table 4.7.

Table 4.8: Comparison between Linux-WSL and Docker (20000-step simulation).

Metric	Linux-WSL	Docker	Difference
Custom factorization	3.80 ms	3.80 ms	$\approx 0\%$
Custom solution	3.79 ms	3.37 ms	-11.1%
Custom solver total	5.73 ms	5.30 ms	-7.5%
PARDISO factorization	1.88 ms	2.02 ms	+7.4%
PARDISO solution	0.26 ms	0.28 ms	+7.7%
PARDISO solver total	1.21 ms	1.29 ms	+6.6%
Assembly time	2.67 ms	3.17 ms	+18.7%
Simulation time (Custom)	429.6 s	418.3 s	-2.6%
Simulation time (PARDISO)	239.0 s	256.3 s	+7.2%

The numerical kernels exhibit remarkably similar behavior in both environments. Differences generally remain below 10%, which is well within the expected variability introduced by operating-system scheduling, memory management, background services and measurement noise. The only component showing a more noticeable degradation is the stiffness matrix assembly stage, which becomes approximately 15–20% slower inside the container.

Overall, the Docker experiments confirm that the proposed optimizations are robust with respect to the execution environment.

4.3.4. Windows Native Evaluation

To complete the performance assessment, the same benchmark campaign was executed directly on the native Windows 11 environment described previously. Due to the lack of compatibility of the adopted MUMPS build with the Windows platform, only the original custom LU solver and the MKL-PARDISO implementation were considered. Table 4.9 summarizes the average timing statistics obtained from the four benchmark cases.

Table 4.9: Average timing statistics on native Windows.

Quantity	**Custom LU** [ms]	**MKL-PARDISO** [ms]	**Variation** [%]
Factorization	4.78	6.07	+27.0
Solution	2.74	0.27	-90.1
Total FEM Solver	5.15	3.32	-35.5
Matrix Assembly	3.01	3.05	+1.3

The results confirm the advantages already observed on Linux-based systems. The most significant improvement is obtained during the solution phase, where MKL-PARDISO reduces the average execution time by approximately one order of magnitude. The total FEM solver time is reduced by about 35%, leading to a visible reduction of the overall simulation time.

However, unlike the Linux-WSL and Docker results, the factorization phase exhibits a considerably larger computational cost. While under Linux MKL-PARDISO achieved factorization times close to 2 ms per call, on Windows the same operation requires approximately 6 ms, making it even slower than the custom solver. This behaviour is consistent with observations reported in the literature and by Intel MKL users. Sparse direct solvers such as PARDISO are highly sensitive to memory allocation policies, thread scheduling, cache locality and runtime libraries[11][5]. Linux environments generally provide lower threading overhead and more efficient memory management for HPC workloads. Since the solution phase still exhibits the expected speedup, the increased factorization cost is likely related to operating-system-level overhead rather than to the numerical algorithm itself. The impact of this behaviour becomes evident when considering the overall simulation times reported in Table 4.10.

Table 4.10: Total simulation times on native Windows.

Time Steps	Custom LU [s]	MKL-PARDISO [s]	Speedup
1000	19.39	18.07	6.8%
5000	90.66	82.65	8.8%
20000	368.05	328.15	10.8%

Compared with Linux-WSL, the gains remain significant but noticeably smaller. For reference, the corresponding Linux-WSL test on 20000 timensteps produced speedups ranging approximately 45%.

Table 4.11: Comparison between Linux-WSL and native Windows results.

Metric	Linux-WSL	Windows
MKL factorization time	$\sim$ 2.0 ms	$\sim$ 6.1 ms
MKL solution time	$\sim$ 0.28 ms	$\sim$ 0.27 ms
MKL total solver time	$\sim$ 1.28 ms	$\sim$ 3.32 ms
Speedup 20K iterations	$\sim$ 44%	$\sim$ 11%

A particularly interesting observation emerges from the comparison of the solution phase. The average solution time remains essentially unchanged across operating systems, indicating that the numerical kernels responsible for the forward and backward substitutions maintain their efficiency. The performance degradation is therefore concentrated almost entirely within the factorization stage.

4.3.5. Overall Comparison on a Long Simulation

To provide a comprehensive assessment of the effectiveness of the implemented optimizations, a final benchmark was performed considering a representative long-duration simulation corresponding to approximately 1000 s of physical time and about 50 500 time steps. All tests were executed on the same hardware described before. The reference configuration corresponds to the original codebase compiled in x86 mode without any optimization. Table 4.12 summarizes the main performance indicators collected during

the benchmark on a windows-native machine, Table 4.13, refers to the results obtained on the Linux-native one.

Table 4.12: Final comparison of all tested implementations for a simulation of approximately 1000 s physical time (50500 time steps).

Configuration	**Solution [ms]**	**Factorization [ms]**	**Solver [ms]**	**Total [min:s]**	**Gain [%]**
Original code (x86, no optimizations)	–	–	–	29:17	–
Custom LU (Windows x64)	2.66	4.70	5.03	16:12	44.7
Custom LU (Linux / WSL)	3.40	3.89	5.38	17:42	39.6
MKL-PARDISO (Windows x64)	0.29	6.55	3.57	14:10	51.6
MUMPS (Linux / WSL)	0.50	6.09	3.55	15:02	48.8
MKL-PARDISO (Linux / WSL)	0.32	2.33	1.48	11:26	61.1

Table 4.13: Performance comparison on native Ubuntu 24.04.4 LTS and (50500 time steps).

Configuration	Sol. [ms]	Fact. [ms]	Solver [ms]	Total [min:s]	Gain [%]
Custom LU (baseline)	5.84	4.45	8.10	23:38	–
MKL-PARDISO	0.38	2.30	1.54	13:05	44.6

The Linux implementation of MKL PARDISO provides the best overall performance. While maintaining a solution time comparable to the Windows version, the average factorization time decreases from 6.55 ms to 2.33 ms, corresponding to a reduction of approximately 64%. This improvement directly translates into a substantial decrease in the average solver time, which reaches only 1.48 ms per call. The overall simulation time is reduced to approximately 11 minutes and 26 seconds, yielding a total speedup of more than 61% with respect to the original implementation. These results indicate that, for the considered application, the numerical solution phase is no longer the primary performance bottleneck. Table 4.13 confirms the trends previously observed in the WSL environment, the gain is only 44.6%,only because as a baseline we refer to the already optimized custom solver.

4.4. Code Profiling

A detailed performance analysis was conducted using CPU profiling tools in order to identify the main computational bottlenecks across different solver implementations. The profiling was carried out for three configurations: the custom LU-based solver (baseline), the same solver coupled with an external sparse direct solver library (MUMPS), and an optimized implementation based on the MKL-PARDISO library.

The goal of this analysis is not only to quantify performance differences, but also to understand how the computational workload is redistributed across the main stages of the finite element workflow when different linear algebra backends are used.

4.4.1. CPU Utilization Analysis

CPU profiling was performed to identify the dominant computational components of each solver configuration and to evaluate how the workload distribution changes when replacing the original linear solver.The baseline implementation is primarily dominated by the linear system solution process.

- Linear system solution: **42.8%**
- Matrix factorization: **29.8%**
- Structural matrix assembly: **16.5%**
- Aerodynamic load computation: **9.7%**

The linear solver accounts for approximately **72.6%** of the total execution time, confirming that the solution of the algebraic system represents the main computational bottleneck in the original implementation.

For what concern the `MUMPS Solver`, after replacing the custom LU implementation with MUMPS, the workload distribution changes significantly.

- Sparse solver routines (MUMPS): **25.5%**
- Sparse factorization driver: **21.9%**
- BLAS kernels (`DGEMM`, `DTRSM`): **12.5%**
- Structural matrix assembly: **43.2%**
- Aerodynamic load computation: **27.7%**

Compared with the baseline configuration, the explicit cost of the custom solution phase is removed and the Computational effort is transferred to sparse factorization and matrix reordering operations. Optimized BLAS kernels become visible among the dominant hotspots, moreover the assembly and physics-related computations represent a larger fraction of the total execution time. The results indicate that MUMPS reduces the cost of the linear solution stage but introduces additional overhead associated with sparse matrix management and factorization.

Looking finally to the `MKL-PARDISO Solver`, the implementation produces the largest shift in workload distribution.

- Structural matrix assembly: **51.4%**
- Aerodynamic load computation: **33.7%**
- Inflow and aerodynamic residual evaluation: **24.3%**
- MKL-PARDISO factorization kernels: **2.7%**
- MKL-PARDISO pivoting and numerical kernels: **1.8%**

Compared with the baseline The solver contribution decreases from approximately **72.6%** to less than **5%** of the total CPU samples, so the Matrix factorization is no longer a dom-

inant cost. Finite element assembly becomes the primary bottleneck, and Aerodynamic or physics-related computations account for a significantly larger share of the execution time.

These results reveal a progressive shift of the computational bottleneck and onfirm that the adoption of optimized sparse direct solvers successfully removes the linear algebra stage as the primary performance limitation.

1. **Custom LU**: solver-dominated execution ($\approx 72.6\%$).
2. **MUMPS**: sparse factorization-dominated execution with significant library overhead.
3. **MKL-PARDISO**: assembly-dominated execution, with the linear solver contributing only a small fraction of the total CPU time.

4.4.2. Cache Misses Analysis

The cache-miss profiling highlights significant differences in memory-access behavior among the three linear solver implementations. While the overall number of cache misses is of the same order of magnitude, the distribution of misses across the execution phases reveals important differences in memory locality and data-access efficiency:

- **Custom CpLambda LU solver:** provide **255.5 million** Total cache misses (lowest among the tested solvers), these misses are distributed across several phases rather than concentrated in a single kernel. The largest contribution originates from the *back-substitution phase* (**22.7%**) and significant contributions are also observed in:
  - memory initialization and array clearing (**13.5–14.3%**),
  - matrix assembly (**11.8%**),
  - factorization-related operations (**8.6%**),
  - matrix conditioning (**4.2%**),
  - sparse factorization (**3.4%**).

  The relatively balanced distribution suggests that cache misses are generated by repeated accesses to sparse matrix structures throughout the solution process rather than by a single dominant kernel.
- **MUMPS solver:** the total cache misses are **358.9 million** (highest among the tested solvers) and nearly half of all cache misses (**48.5%**) occur inside the sparse direct solver. The most significant solver-related contributions are:
  - elimination-tree and frontal-matrix factorization kernels (**17.4%**),
  - assembly and symbolic processing kernels (**7.7%**),
  - forward/backward solve stages (**7–10%**).

Memory initialization still accounts for a large fraction of misses (**15.1%**) and the concentration of misses inside the solver indicates that the multifrontal factorization algorithm requires intensive access to large temporary data structures, reducing cache locality.

- **MKL/PARDISO solver:** here we have **305.4 million** cache misses, more concentrated than in the custom solver but less than in MUMPS. The largest contribution comes from the sparse factorization kernel (**18.8%**). Additional solver-related hotspots include:
  - dense matrix multiplication kernels (**4.0%** and **3.1%**),
  - matrix-vector operations (**2.3%**),
  - block triangular solves (**1.1%**),
  - pivoting and factorization support routines (**1.3%**).

  Memory initialization remains a major source of cache misses (**15.0%**). The presence of highly optimized BLAS kernels suggests that cache misses are concentrated in computationally intensive blocked operations, which are designed to maximize data reuse despite operating on large datasets.

Table 4.14: Cache-miss comparison of the tested solvers.

Solver	Misses [M]	Main hotspot	Pro
Custom LU	255.5	Back-subst. (22.7%)	Best locality
MUMPS	358.9	Multifrontal fact. (48.5%)	Robust sparse solver
MKL/PARDISO	305.4	Factorization (18.8%)	Optimized BLAS kernels

The custom LU implementation exhibits the lowest overall number of cache misses (**255.5M**), indicating better memory locality for the problem size considered. The MKL/PARDISO implementation generates approximately **19.5%** more cache misses (**305.4M**), while MUMPS produces approximately **40.4%** more cache misses (**358.9M**). The custom solver spreads memory traffic across several stages of the algorithm, whereas both external solvers concentrate cache misses within highly optimized factorization and solve kernels. This behavior reflects the different algorithmic strategies adopted by the three approaches and highlights the memory-intensive nature of multifrontal and blocked sparse direct methods[1][15][7]. Finally, analysis conducted on previous version of the code shows that all the optimizations made to reduce the cash misses for the stiffness matrix assembly produced negligible results in terms of improving the performances.

4.4.3. Memory Analysis

To evaluate the memory footprint of the different solver implementations, all three versions of the code were profiled using `heaptrack`. The analysis focused on three complementary metrics: consumed memory, cumulative memory allocations, and temporary memory allocations.

Consumed Memory The memory consumption over time is reported in Figure 4.11. All three implementations exhibit a very similar behavior, characterized by an initial peak of approximately 180MB. This peak is not related to the linear solvers themselves, but rather to the current implementation of the matrix-format conversion routine. During the conversion from the native skyline storage format to CSR, CSC, or coordinate formats, an intermediate dense matrix of size `ndof` $\times$ `ndof` is temporarily allocated and initialized to zero before being immediately deallocated. This implementation is clearly suboptimal and represents a potential target for future optimization.

After the initialization phase, the memory usage stabilizes and remains approximately constant throughout the simulation, corresponding to about 1000 time steps (90s of execution time):

- **Custom CpLambda LU solver:** approximately 30MB;
- **MUMPS solver:** approximately 40MB;
- **MKL/PARDISO solver:** approximately 60MB.

The nearly constant memory consumption observed after the initial transient indicates the absence of memory leaks in all three implementations.

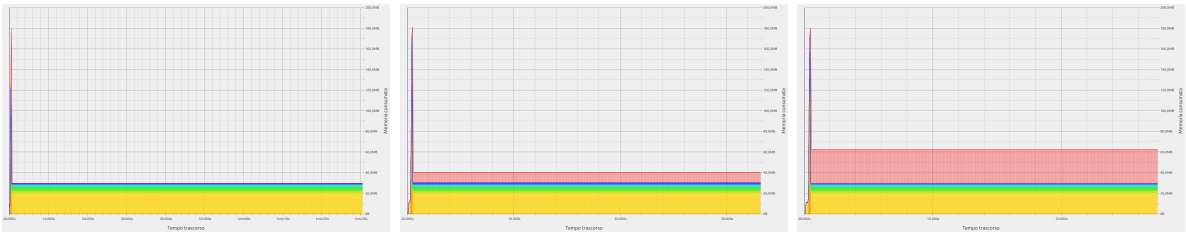

Figure 4.11: Memory consumption timeline CpLambda - MUMPS - MKL

Cumulative Memory Allocations The cumulative number of memory allocations is shown in Figure 4.12. As expected, all curves exhibit a linear trend, since the metric represents the accumulated number of allocations during the simulation.

A more detailed analysis reveals significant differences among the three approaches:

- **Custom CpLambda LU solver:** Most allocations occur during initialization and matrix-format conversion. Once the simulation starts, the allocation rate remains nearly constant, the only noticeable growth is associated with dynamic vector management, string handling, and file I/O operations. The total number of allocations increases from approximately 1.6×10^4 to 1.8×10^4 during the entire simulation.
- **MUMPS solver:** The baseline allocations related to the application itself are comparable to those of the custom solver, however the routines belonging to the `dmumps` library introduce a very large number of additional allocations. The cumulative allocation count reaches approximately 1.5×10^5 at the end of the simulation.
- **MKL/PARDISO solver:** The allocation pattern is very similar to that of the custom solver, the main additional cost arises from the allocation of an intermediate

solution vector of size `ndof` at each time step. Despite this non-optimal implementation, the MKL internal routines contribute only marginally to the overall allocation count. The final number of allocations remains below 2.5×10^4.

The significantly larger number of allocations generated by MUMPS is likely one of the factors contributing to its slower factorization stage. This observation is consistent with the CPU profiling results, where the factorization phase represents the dominant computational bottleneck, while the solution phase remains competitive with the MKL implementation.

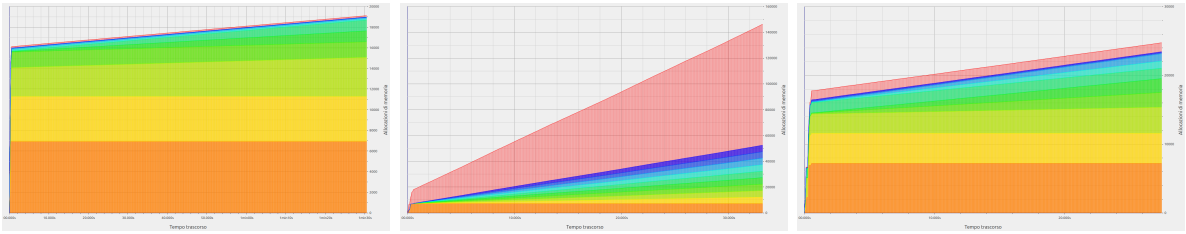

Figure 4.12: Memory allocation timeline CpLambda - MUMPS - MKL

Temporary Memory Allocations Figure 4.13 reports the cumulative number of temporary memory allocations generated during execution. As for the previous metric, all curves display a linear growth over time.

- **Custom CpLambda LU solver:** Approximately 10^3 temporary allocations are generated during the simulation, nearly all of them originate from `libc.so.6` and are associated with file I/O operations.
- **MUMPS solver:** The total number of temporary allocations reaches approximately 10^4. File I/O operations account for only about 10% of the total, the remaining allocations are generated internally by the MUMPS numerical routines.
- **MKL/PARDISO solver:** Approximately 2×10^3 temporary allocations are observed. Most of the additional allocations are associated with the `mkl_internal_malloc` routines provided by the Intel OneAPI runtime.

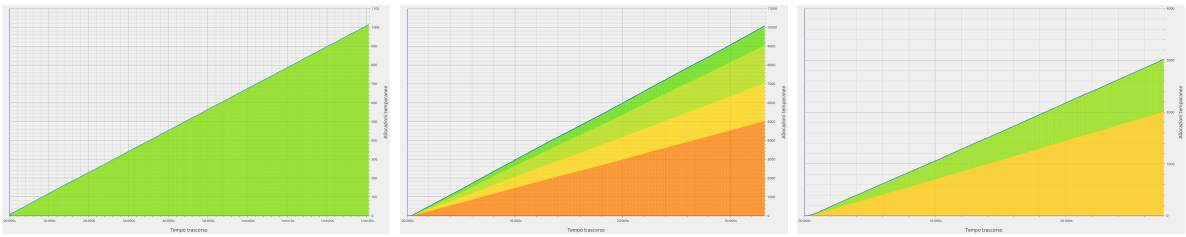

Figure 4.13: Temporary memory timeline CpLambda - MUMPS - MKL

The memory profiling results further support the conclusions drawn from the CPU and cache analyses. The custom LU solver achieves the lowest memory footprint but lacks the advanced numerical optimizations available in modern sparse direct solvers. MUMPS

introduces a very large number of dynamic allocations, which likely contributes to the increased factorization cost observed in the performance analysis. In contrast, MKL/PARDISO maintains a relatively low allocation count while leveraging highly optimized numerical kernels.

5 | Conclusions

5.1. Final Results

Solver	Advantages	Drawbacks
Custom LU	• Lowest memory footprint • Limited number of memory allocations • Full control over implementation details • Native integration within the framework	• Skyline storage unsuitable for large sparse systems • High factorization time • Significant cache-miss contribution during factorization • Limited exploitation of modern hardware optimizations
MUMPS	• Robust and mature solver • Suitable for very large sparse systems • Advanced multifrontal factorization algorithms	• Higher memory consumption • Large number of dynamic allocations • Significant factorization overhead • Sensitive on parameters optimization
MKL Pardiso	• Best overall performance • Efficient cache utilization • Limited allocation overhead • Optimized BLAS backend • Win & Linux libraries available	• Dependence on proprietary libraries • Limited control over internal algorithms

Table 5.1: Summary of the main advantages and drawbacks of the investigated solvers.

The benchmark campaign and the profiling analyses (execution time, CPU usage, cache behavior, and memory consumption) provide a comprehensive overview of the strengths and limitations of the three investigated solution strategies. Table 5.1 summarizes the main findings.

Based on all the performed analyses, the integration of Intel MKL Pardiso emerges as the most effective solution for the considered application. The investigated structural problems are sufficiently large to benefit from advanced sparse direct solvers, but not large enough to fully exploit the distributed-memory capabilities of multifrontal approaches[2].

MKL Pardiso employs a supernodal factorization strategy that maps efficiently onto optimized BLAS kernels and modern cache hierarchies[8]. The relatively narrow bandwidth and regular sparsity pattern of the analyzed matrices generate predictable fill-in structures, allowing the supernodal approach to achieve high computational throughput while maintaining a moderate memory footprint[10]. Furthermore, MKL extensively leverages hardware-specific optimizations such as vectorization, cache-aware blocking, and multithreading, resulting in lower execution times and reduced memory overhead[11].

Conversely, MUMPS is primarily designed for large-scale distributed-memory environments. Its multifrontal algorithm provides greater flexibility for highly irregular sparse patterns and can become advantageous when dealing with significantly larger systems or memory-constrained cluster architectures[12][13]. However, for the problem sizes considered in this work, the additional memory allocations and factorization overhead outweigh the potential benefits.

Therefore, among the investigated alternatives, MKL Pardiso represents the most advantageous compromise between computational performance, memory efficiency, and integration effort for the current Cp-Lambda framework.

5.2. Final Considerations

Modern computing technologies provide several opportunities for improving the performance of large-scale numerical simulations. Parallel architectures, multicore processors, and GPU accelerators have significantly expanded the range of available optimization strategies. At the beginning of this work, the objective was to exploit these technologies as much as possible in order to achieve substantial reductions in computational time. However, the practical limitations imposed by an existing legacy codebase proved to be more restrictive than initially anticipated.

The first challenge was related to the nature of the problem itself. The structural models considered in this work generate sparse stiffness matrices that are large, but not extremely large, and characterized by a relatively narrow bandwidth and poor conditioning. These characteristics naturally favor the use of direct solvers over iterative approaches.

Although modern GPU technologies can provide remarkable acceleration for iterative methods, their effective application typically requires suitable preconditioners and additional memory management strategies. The associated implementation complexity, memory overhead, and reduced portability would likely outweigh the potential performance

gains for the class of problems investigated here. Consequently, GPU-based iterative approaches were not considered a practical solution for the current framework. Different considerations may apply to other computationally intensive stages, such as the assembly of the structural matrices, where a higher degree of parallelism could potentially be exploited in future developments.

For these reasons, the optimization effort focused on modern sparse direct solvers. The integration of well-established numerical libraries proved to be a more effective strategy than attempting to redesign the solver infrastructure from scratch. Among the investigated alternatives, Intel MKL Pardiso provided the most favorable balance between computational efficiency, memory usage, and implementation effort.

Overall, the developed solution achieved an execution time reduction of approximately 65% with respect to the original x86 implementation. While this result represents a significant improvement, it remains below the expectations initially associated with the adoption of modern hardware and numerical libraries. The analysis revealed that a considerable portion of the development effort was not devoted to the implementation of new numerical algorithms, but rather to understanding the existing codebase, resolving compatibility issues, debugging legacy components, and enabling a stable x64 compilation environment capable of exploiting modern optimized libraries.

This observation naturally leads back to the original question that motivated the work:

> *Is it preferable to modernize a legacy software framework or to rewrite it from scratch?*

As is often the case in engineering, the answer is: *it depends*.

For the specific application considered in this study, modernizing the existing framework proved to be the most appropriate choice. The computational cost of the simulations remains in the order of several hours, making performance improvements highly valuable, while the absence of strict real-time constraints allowed the modernization process to be carried out incrementally.

Perhaps the most important advantage of preserving the existing framework lies beyond the performance gains themselves. Reusing the established data structures, numerical workflow, and convergence management mechanisms made it possible to maintain the reliability accumulated over years of development. Moreover, retaining the original implementation enabled direct comparisons between different solvers using identical benchmark models, significantly simplifying the validation process.

From this perspective, the main outcome of this work is not merely the integration of faster numerical solvers, but the demonstration that substantial performance improvements can be achieved while preserving the robustness, validation history, and domain-specific knowledge embedded within a mature engineering software framework.

5.3. Future Developments

One of the most important outcomes of the present study is that the primary computational bottleneck has been successfully shifted from the numerical solver to the stiffness matrix assembly phase. While the solver infrastructure has been extensively analyzed and optimized through the integration of modern sparse direct solvers, the assembly procedures themselves have remained essentially unchanged.

In the current implementation, the update of structural and aerodynamic entities is largely performed sequentially. However, these operations are inherently parallel in nature, as many element- and node-level computations can be executed independently. Consequently, the matrix assembly process represents the most promising area for future performance improvements and may potentially benefit from both shared-memory and distributed-memory parallelization strategies.

Furthermore, several aspects of the current implementation still deserve additional investigation:

- Further tuning of the numerical parameters and internal options available in both MUMPS and MKL Pardiso in order to identify solver configurations specifically optimized for the sparsity patterns generated by the Cp-Lambda framework.
- Redesign of the skyline-to-sparse conversion routines, which currently generate avoidable memory allocation peaks due to the temporary creation of intermediate dense structures.
- Reduction of residual software inefficiencies, including redundant memory allocations, unnecessary file I/O operations, and repeated function calls that still contribute to the overall execution time.
- Development of a parallel matrix assembly strategy capable of exploiting the intrinsic independence of many element-level computations, potentially representing the next major step in reducing simulation times.
- Evaluation of the distributed-memory capabilities of MUMPS through MPI-based implementations, with particular attention to scalability on multi-node architectures and larger problem sizes.

The results presented throughout this work suggest that the greatest opportunities for future improvements are no longer associated with the numerical solution phase itself, but rather with the surrounding computational infrastructure. In this respect, the modernization effort has successfully established a more efficient solver backend while simultaneously identifying the next set of performance-critical components that should be addressed in future developments.

5.4. Considerations on the Use of AI-Assisted Development Tools

The rapid evolution of Large Language Models (LLMs) and AI-assisted development tools has significantly influenced software engineering workflows in recent years. Since such tools were extensively employed throughout the development of this work, it is worth briefly discussing their practical impact on the project.

The observations reported in this section are based exclusively on the author's experience during the development of the thesis and should not be interpreted as a rigorous scientific evaluation. No formal benchmarks were conducted, and the conclusions are not derived from published literature. The tools employed included models developed by Anthropic, Google, and OpenAI, without the use of autonomous multi-agent systems.

Several strengths emerged consistently throughout the project:

- Rapid and generally reliable code analysis.
- Efficient generation of simple utility functions.
- Effective assistance in resolving compilation, linking, and build-system issues.
- Fast navigation and interpretation of large codebases.
- Generation of detailed reports describing software architecture and execution flow.

In particular, AI tools proved extremely useful when working with a large legacy codebase composed of thousands of source files. Tasks that would normally require several days of manual inspection, such as identifying the role of specific modules, tracing function dependencies, or reconstructing execution workflows, could often be completed within minutes. Similar benefits were observed during the migration to the x64 environment, where AI-assisted analysis frequently helped identify the origin of compilation and linking errors involving external libraries, build configurations, and dependency mismatches.

One notable example concerned the integration of MUMPS, where linking issues originated from incompatibilities between different runtime library configurations. In such cases, AI tools often accelerated the diagnosis process considerably, even when the underlying problem remained technically complex.

At the same time, several limitations became evident:

- Limited reliability of proposed solutions without independent verification.
- Tendency to increase complexity in code sections that would benefit from simplification.
- Poor effectiveness when debugging numerically incorrect results.
- Limited reliability when contributing to complex numerical algorithms.
- Strong dependence on the quality and precision of the provided prompts.

More generally, these systems should be viewed as statistical assistants rather than autonomous software engineers. Their effectiveness was found to be strongly correlated with the user's understanding of both the problem and the codebase. When employed to perform well-defined and narrowly scoped tasks, the generated results were often useful and time-saving. Conversely, when asked to operate autonomously on larger portions of complex numerical software, the proposed solutions frequently became misleading, introducing additional complexity or addressing problems that did not actually exist.

A similar distinction was observed during debugging activities. While AI tools were often effective at diagnosing compilation failures and configuration issues, they proved considerably less reliable when investigating bugs that produced numerically incorrect but syntactically valid results. In these situations, the generated hypotheses often lacked sufficient understanding of the physical and numerical context required to identify the true source of the problem.

The overall experience suggests that AI-assisted development tools are most valuable when used to accelerate tasks that have already been clearly identified by the developer. Their greatest contribution lies in reducing the time required for code exploration, documentation, boilerplate generation, and build-system troubleshooting. Conversely, they appear significantly less effective when expected to independently discover the root causes of complex numerical or architectural issues.

Ultimately, the effectiveness of these tools depends heavily on the expertise of the user. In the context of this work, AI assistance often produced substantial time savings, but in other situations it generated misleading solutions that required significant effort to identify and correct. As a result, the net benefit was highly task-dependent, reinforcing the view that these systems currently function best as productivity amplifiers rather than substitutes for domain expertise.

Bibliography

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] P. R. Amestoy et al. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 2002. URL `https://dl.acm.org/doi/10.1145/504210.504212`.

[3] O. A. Bauchau, C. L. Bottasso, and Y. G. Nikishkov. Modeling rotorcraft dynamics with finite element multibody procedures. *Mathematical and Computer Modelling*, 33(10–11):1113–1137, 2001. doi: 10.1016/S0895-7177(00)00303-4.

[4] O. A. Bauchau, C. L. Bottasso, and L. Trainelli. Robust integration schemes for flexible multibody systems. *Computer Methods in Applied Mechanics and Engineering*, 192(3–4):395–420, 2003. doi: 10.1016/S0045-7825(02)00519-4.

[5] J. D. Booth, S. Rajamanickam, and H. Thornquist. Basker: A threaded sparse lu factorization utilizing hierarchical parallelism and data layouts. *SIAM Journal on Scientific Computing*, 2016.

[6] C. L. Bottasso, A. Croce, B. Savini, W. Sirchi, and L. Trainelli. Aero-servo-elastic modeling and control of wind turbines using finite-element multibody procedures. *Multibody System Dynamics*, 16(4):291–308, 2006. doi: 10.1007/s11044-006-9027-1.

[7] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[8] M. Ferrari. A comparison of sparse solvers for severely ill-conditioned linear systems in geophysical marker-in-cell simulations. *arXiv preprint arXiv:2409.11515*, 2024. doi: 10.48550/arXiv.2409.11515. URL `https://arxiv.org/abs/2409.11515`. Version 2, last revised 23 Sep 2024.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4 edition, 2013.

[10] Intel Corporation. *Developer Guide for Intel oneAPI Math Kernel Library for Linux*, 2024. URL `https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2024-2/overview.html`.

[11] Intel Corporation. *Intel oneAPI Math Kernel Library Developer Reference: PARDISO Parallel Direct Sparse Solver Interface*, 2024. URL `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-fortran`.

[12] MUMPS team. Mumps documentation, 2025. URL https://mumps-solver.org/index.php?page=doc.

[13] MUMPS team. *MUMPS Users' Guide*, 2025. URL https://mumps-solver.org/doc/userguide_5.8.0.pdf.

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[15] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.