Dependency-Aware Scheduling for Parallel Nonlinear System Solving on Multicore Architectures

Adel Moussaoui[1,*], Foudil Belhadj[2], El Yazid Gueddoudj[3], Salim Bouamama[4] and Nawel Boukhari[5]
[1]Laboratory of Informatics and its Applications of M'sila (LIAM), Faculty of Mathematics and Computer Science, University of M'sila, PO Box 166 Ichebilia, 28000, M'sila, Algeria
[2]LMSE Laboratory, Mohamed El Bachir El-Ibrahimi University, Bordj Bou Arréridj, Algeria
[3]Institute of Architecture and Urbanism, University of Batna1, Batna, Algeria | LRIT laboratory, University of Tlemcen, Tlemcen, Algeria
[4]Mechatronics Laboratory (LMETR), Optics and Precision Mechanics Institute, University Setif 1 - Ferhat Abbas, Setif, 19000, Algeria | Department of Computer Science, University Setif 1 - Ferhat Abbas, Setif, 19000, Algeria
[5]University of M'sila, Department of Computer Science, PO Box 166 Ichebilia, 28000 M'sila, Algeria
E-mail: adel.moussaoui@univ-msila.dz
[*]Corresponding author

Keywords: Nonlinear systems of equations, task scheduling, DAG algorithms, parallel computing, multicore architectures, critical path method, dependency-aware decomposition

Received: August 12, 2025

This paper presents a comprehensive experimental evaluation of ten Directed Acyclic Graph (DAG) scheduling algorithms for the parallel solution of decomposed nonlinear equation systems on multicore processors. Because parallel execution fundamentally depends on system decomposition, we abstract the decomposition stage and model the resulting task structures as DAGs. This abstraction enables systematic evaluation across diverse dependency patterns without restricting the study to a specific decomposition technique. We compare critical-path-aware, heuristic-based, and baseline schedulers across sparse, balanced, and dense DAGs on a multicore platform with 1, 2, 4, and 8 cores using real polynomial systems with $O(n^3)$ computational complexity. Results show that structural decomposition alone yields speedups of 65–185$\times$ over monolithic solving, while parallel scheduling further amplifies these gains by 1.13–1.53$\times$, producing total accelerations of 74–283$\times$ and reducing multi-hour computations to practical runtimes on commodity multicore hardware. No single scheduler consistently dominates across all structural regimes. Parallel efficiency is strongly dependency-dependent: sparse graphs scale effectively up to 4 cores, whereas balanced and dense graphs exhibit limited scalability beyond 2 cores as structural coupling and shared-memory constraints restrict available concurrency. These findings indicate that optimal performance requires aligning scheduler selection and core allocation with the dependency structure induced by decomposition.

Povzetek: Članek eksperimentalno primerja deset algoritmov za razporejanje nalog v usmerjenih acikličnih grafih pri vzporednem reševanju razčlenjenih nelinearnih sistemov na večjedrnih procesorjih. Rezultati kažejo velike pohitritve, vendar je učinkovitost odvisna od strukture odvisnosti, zato je treba izbiro algoritma in števila jeder prilagoditi posameznemu grafu.

1 Introduction

Systems of equations are widely used in engineering simulations and software tools across many fields, making them a fundamental tool for analyzing and modeling complex systems in engineering and scientific research [1]. Nonlinear equations are essential for modeling complex phenomena that cannot be described by simple linear relationships [2]. For instance, in aerospace engineering, nonlinear equations are used to model fluid flow around aircraft and spacecraft, including the study of turbulence, supersonic and hypersonic flow, and shock waves [3]. In mechanical engineering, nonlinear equations help model the behavior of mechanical systems subject to large deformations or vibrations [4]. They are also used in studying materials with nonlinear mechanical properties, such as rubber and polymers [5] or thermal-hydraulic. In economics, nonlinear equations model market behavior, production and consumption decisions, and macroeconomic dynamics [6].

These are just a few examples of engineering fields that rely on nonlinear equations. In general, any engineering system that involves complex interactions or nonlinear behavior can benefit from their use [7]. Therefore, developing effective and efficient methods for solving nonlinear equations is crucial, especially given that these solutions are typically computed on workstations with multicore CPUs.

A system of equations is a set of two or more equations with multiple variables, where the goal is to find values for the variables that satisfy all the equations simultaneously. We can write a system of equations as:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases}$$

Where $f_1, f_2, \ldots, f_n$ are functions that define relationships among the variables $x_1, x_2, \ldots, x_n$. If these functions are linear, the system is called a **linear system**; if they are nonlinear, it is a **nonlinear system**.

The most used solving methods of nonlinear systems are the algebraic methods, which are a class of numerical methods. They work by generating a sequence of approximate solutions that converge to the exact solution of the system [8]. They start with an initial estimate and use an iterative process to repeatedly refine the solution until a desired level of accuracy is achieved. The resolution time depends on the complexity of the solving method and the number of repetitions required for the solution [9]. There are many numerical solvers available for systems of nonlinear equations; we can find them in many software libraries, each with their own strengths and weaknesses [10]. One common approach is to use iterative methods, such as Newton's method [8] or the Broyden-Fletcher-Goldfarb-Shanno method [11], which approximate the solution by successively refining an initial guess. Other methods include the secant method [12] and the quasi-Newton method [11]. In addition, there are specialized solvers designed for specific types of nonlinear systems, such as the Levenberg-Marquardt method for least-squares problems [13] or the homotopy continuation method [14] for solving polynomial systems. The time complexity of those solving methods depends on several factors, including the dimension of the system of equations, the desired accuracy of the solution, and the required initial guess [15]. In general, the best of those solving methods has a convergence rate of quadratic order, which means that the number of correct digits in the solution doubles with each iteration [16]. The time complexity of each iteration is $\mathcal{O}(n^3)$ [17], where n is the dimension of the system. Therefore, the total time complexity is typically $\mathcal{O}(kn^3)$, where k is the number of iterations required to achieve the desired accuracy. The larger the value of k the more precise the solution will be. As the number of variables increases, the solving time also increases, resulting in longer computation times. Solving a system of more than one hundred equations is a challenge, and any method that speeds up the process is valuable [18].

1.1 A scheduling strategy for parallel subsystem solutions

When systems of equations can be decomposed into subsystems with dependency relationships, the resolution process becomes amenable to parallel execution. The efficiency of such parallel approaches critically depends on how subsystem evaluations are scheduled across available computational resources. This scheduling problem—determining which subsystem to execute on which processor at what time—has been extensively studied in the parallel computing literature under the framework of Directed Acyclic Graph (DAG) scheduling [40, 37].

DAG scheduling algorithms can be broadly classified into three categories. **Critical path-aware schedulers**, such as HEFT (Heterogeneous Earliest Finish Time) [38] and CPOP (Critical Path on a Processor) [38], explicitly compute the longest execution path through the dependency graph and prioritize tasks along this path to minimize overall makespan. **Heuristic-based schedulers** employ simpler strategies based on task properties or graph structure, such as processing tasks level-by-level or prioritizing by size, without explicit critical path computation [37]. **Baseline schedulers**, such as FIFO or random selection, provide performance bounds with minimal computational overhead.

While critical path-aware schedulers are widely recommended in literature for their theoretical optimality in heterogeneous environments [38], their practical advantage over simpler heuristics in homogeneous multi-core systems—the most common deployment scenario for scientific workstation computing—remains poorly quantified. Most prior comparative studies focus on heterogeneous clusters with varying processor speeds [42, 41] or use synthetic task costs that may not reflect real computational complexity [40]. This leaves a significant gap in understanding which scheduling strategies are most effective for real scientific computing workloads executed on contemporary homogeneous multi-core processors.

1.2 Research objectives and contributions

Solving large systems of nonlinear equations can be computationally expensive. Ait Audia et al. [35] demonstrated that decomposing the system into smaller subsystems using the Dulmage-Mendelsohn partitioning method [36] provides significant acceleration by reducing task size and exposing dependency relationships. In this work, we introduce parallelization as a complementary approach, distributing the decomposed subsystems across multiple cores to enable simultaneous resolution. We systematically evaluate multiple DAG scheduling strategies on homogeneous multi-core processors, investigating how core count affects speedup, efficiency, and resource utilization under different workload structures and dependency patterns. The primary objective of this work is to systematically quantify the combined impact of structural decomposition and parallel scheduling, and to analyze how workload structure, scheduling policy, and core count jointly determine scalability, efficiency, and practical performance limits in homogeneous multi-core environments. Our method accelerates nonlinear system resolution at the structural and scheduling levels enabled by system decomposition, without mod-

ifying the underlying numerical solvers. Subsystems are processed using standard solvers, while optimized parallel execution and scheduling reduce overall computation time in a solver-agnostic and non-intrusive manner. The main contributions of this work are summarized as follows:

1. Provide a systematic and comparative evaluation of ten DAG scheduling algorithms spanning three major categories: critical path–aware methods (HEFT, CPOP, ETF, CP_Priority), heuristic-based strategies (level_by_level, largest_first, smallest_first, most_successors_first), and baseline approaches (FIFO, random).
2. Quantify the performance impact of structural decomposition and parallel scheduling using real computational workloads on a representative modern multicore processor.
3. Evaluate scheduler behavior across five distinct DAG topologies capturing varying degrees of parallelism and dependency density.
4. Analyze scaling characteristics across multiple core configurations, identifying efficiency limits and optimal resource allocation strategies.

The implementation of all scheduling algorithms and the benchmarking framework are publicly available at [43]. The experimental results highlight two key findings. First, task decomposition combined with parallel execution yields substantial speedups exceeding two orders of magnitude compared to monolithic execution. Second, simple heuristic schedulers achieve performance comparable to or exceeding that of critical path–aware algorithms in homogeneous environments, challenging the common assumption that critical path awareness consistently guarantees superior performance.

Rather than reimplementing a specific decomposition technique, this work abstracts the decomposition stage by generating dependency graphs that emulate decomposition outputs. This abstraction enables systematic, controlled, and reproducible evaluation of scheduling strategies across diverse structural scenarios, isolating scheduling effects from decomposition-specific artifacts.

2 Related work

The parallelization of numerical algorithms and task scheduling on multi-core and heterogeneous systems has been extensively studied. Existing research can be broadly categorized into approaches focusing on: (1) DAG-based task scheduling and real-time analysis, (2) parallel sparse and dense linear algebra solvers, and (3) the parallelization of other specific numerical methods. This section reviews key contributions in these areas.

A significant body of work models parallel tasks as Directed Acyclic Graphs (DAGs) to exploit parallelism while managing dependencies for real-time systems. A multi-DAG scheduling approach was presented in [19] that prevents inter-task interference by regulating core allocation per DAG, enabling fine-grained concurrent analysis in multi-task scenarios and improving schedulability. In heterogeneous computing systems, knapsack-based co-scheduling approaches have been shown to significantly reduce makespan compared to state-of-the-art methods [20]. Focusing on heterogeneous multi-cores, a criticality-based scheduling algorithm for typed DAG tasks was introduced in [21], where vertex priority depends on remaining workload. The proposed worst-case response time (WCRT) analysis for this non-work-conserving strategy improved schedulability analysis accuracy by an average of 20%. Similarly, a novel DAG task scheduling method for heterogeneous multi-cores is described in [22], where the task graph is reconstructed by adding execution edges to eliminate blocking nodes, reducing pessimism in WCRT analysis and improving accuracy by 20% compared to traditional bounds. More recently, an improved list scheduling algorithm for DAG-based task graphs on multi-core edge servers demonstrated significant latency reduction through structured task prioritization [24].

Efficient parallel solvers for linear systems are crucial for scientific computing. Research in this domain spans direct methods, iterative methods, and task-based frameworks. For **dense direct methods**, an OpenMP pipelined algorithm using a queue structure was developed in [23], demonstrating superior performance over naive parallel approaches along with a practical performance model. The work in [25] evaluates task-based frameworks for dense linear algebra, finding that the SMPSs model, which automatically constructs arbitrary DAGs, provides higher automation and more flexible scheduling than alternatives like Cilk. Research in [26] demonstrates that within asynchronous, data-driven DAG execution environments on multicores, Right-Looking (RL) tile algorithm variants can outperform traditional cache-aware Left-Looking (LL) variants by exposing greater parallelism. A scalable runtime system for the dynamic scheduling of tiled linear algebra algorithms on distributed-memory multicore systems is introduced in [27], featuring a distributed algorithm to resolve data dependencies without communication. For **sparse direct and iterative methods**, the study in [28] designed a portable sparse direct solver for heterogeneous CPU-GPU systems, employing a hybrid computing model with dynamic load balancing to exploit parallelism, achieving significant speed-ups. A modular software framework for sparse linear solvers on hybrid CPU-GPU systems was proposed in [29], separating core sparse matrix management from preconditioner setup. Specific work on **triangular solvers** includes [30], where three scheduling approaches for parallel triangular solvers on multicores were designed and evaluated, identifying the Column-Oriented Scheduling (COS) strategy as the most efficient. A static scheduling approach with load balancing for triangular band systems was proposed in [31] by eliminating unnec-

essary dependencies, achieving results that matched theoretical lower bounds.

Parallel techniques have been applied to various other numerical problems. A parallel Newton's method for solving systems of nonlinear equations was developed in [32], exploiting multiple levels of parallelism on multi-core architectures to reduce execution times. The bottleneck of inter-node communication in parallel ordinary differential equation (ODE) solving is addressed in [33] through an adaptive algorithm combining the Adams–Moulton and Parareal methods, optimized for heterogeneous CPU/GPU resources. Non-intrusive parallelization techniques (parallel linear solvers and OpenMP) for multibody dynamics simulations were evaluated in [34], finding both approaches effective for achieving significant speedups.

While prior work has extensively explored DAG scheduling for real-time guarantees, heterogeneous architectures, and task-based runtime systems, as well as parallel implementations of linear algebra and other numerical solvers, comparatively little attention has been devoted to systematically evaluating multiple scheduling strategies across controlled variations in dependency structure for nonlinear equation systems. Existing studies often focus on improving specific algorithms, tightening worst-case bounds, or optimizing runtime frameworks, rather than examining how scheduler design interacts with structural characteristics induced by system decomposition.

Table 1 summarizes the key methods surveyed above along with their scheduling strategy, target architecture, and primary performance focus, illustrating the structural gaps addressed by this work.

In contrast, this work provides a comprehensive comparative evaluation of diverse scheduling strategies under controlled DAG topologies that emulate decomposed nonlinear systems, with an emphasis on quantifying the combined impact of structural decomposition and parallel scheduling on practical performance. This positioning differentiates our study from prior solver-centric and real-time-centric approaches.

3 Structural modeling and decomposition of nonlinear systems

3.1 Graph-theoretic modeling of nonlinear equation systems

A system of nonlinear equations can be structurally modeled as a bipartite graph $G = (M, N, E)$, where the vertex set is partitioned into two disjoint subsets: M and N [36]. Each vertex in M represents an equation, while each vertex in N represents a variable of the system. An edge $(m_i, n_j) \in E$ exists if and only if the variable n_j appears in equation m_i.

The bipartite graph $G = (M, N, E)$ provides a structured representation of the relationships between equations and variables, facilitating structural analysis of the system. Through this representation, unsolvable or structurally singular components can be identified and isolated. The structurally solvable portion of the system is subsequently decomposed into smaller, more manageable subsystems. An important outcome of this decomposition process is the construction of a dependency graph that captures interdependencies among subsystems. This graph determines the order in which subsystems must be solved and identifies components that can be executed in parallel.

System partitioning is performed using a maximum matching–based approach. First, a perfect matching is identified in the bipartite graph $G = (M, N, E)$ representing the system. Based on this matching, a directed graph G_0 is constructed by replacing each matched edge (m_i, n_j) with two directed arcs (m_i, n_j) and (n_j, m_i), while all unmatched edges are oriented from M to N. The strongly connected components (SCCs) of G_0 are then computed, where each SCC corresponds to an irreducible solvable subsystem. These components are contracted into single vertices to form a reduced graph R, whose directed edges represent dependencies between subsystems. A topological ordering of R yields a valid execution sequence for solving the subsystems.

Because this method relies solely on structural incidence information—specifically, which variables appear in which equations—it is independent of the algebraic form of the equations. Consequently, it can be applied to systems containing polynomial, rational, radical, exponential, logarithmic, or trigonometric expressions. Further details on structural partitioning of equation systems can be found in [35].

Importantly, the reduced dependency graph R is acyclic, ensuring that no circular dependencies exist among subsystems. This property guarantees the existence of a valid solution sequence. Each subsystem can therefore be solved independently according to the topological order prescribed by R. Since solving a single large system is generally more computationally demanding than solving multiple smaller subsystems, the overall computation time is largely determined by the size of the largest subsystem. As a result, even under sequential execution, partitioning can yield computational speedup due to reduced dimensionality and improved numerical efficiency.

3.2 Complexity reduction through partitioning

Many iterative nonlinear solvers, such as Newton–Raphson, require the factorization or inversion of an $n \times n$ Jacobian matrix, typically with computational complexity $\mathcal{O}(n^3)$ [8, 17]. For large-scale systems (e.g., $n = 1{,}000$), this cost becomes substantial. If the system is partitioned into k smaller subsystems, each of size m, the total computational cost is approximately $\mathcal{O}(km^3)$, assuming weak coupling between subsystems [18]. When $m \ll n$, this reduction can significantly decrease overall execution time while preserving solution accuracy.

Table 1: Summary of related scheduling and parallelization methods

Reference	Method	Architecture	Workload	Focus
[19]	DAG modeling	Multicore	Real-time tasks	Schedulability
[21]	Criticality-based	Heterogeneous	Real-time tasks	WCRT analysis
[22]	Graph reconstruction	Heterogeneous	DAG tasks	WCRT reduction
[25]	Task-based frameworks	Multicore	Dense linear algebra	Automation & flexibility
[26]	Dynamic scheduling	Multicore	Tile linear algebra	Parallelism exposure
[30]	Static scheduling	Multicore	Triangular solvers	Efficiency
[32]	Parallel Newton	Multicore	Nonlinear systems	Runtime reduction
This work	**DAG scheduling**	Multicore	**Nonlinear systems**	**Comparative evaluation**

In practice, strongly connected components are identified using Tarjan's linear-time algorithm, which is widely employed for detecting maximal strongly connected subgraphs in directed graphs. Each SCC corresponds to a block of mutually dependent equations and variables that must be solved sequentially. Contracting these SCCs produces a condensation graph that is acyclic by construction. This directed acyclic graph (DAG) serves as the structural foundation for the subsequent parallelization and scheduling stage. The resulting condensation graph constitutes a task-level Directed Acyclic Graph (DAG), where each vertex represents an independent subsystem and each edge encodes a precedence constraint. While the structural decomposition stage determines what can be solved independently, it does not specify how these subsystems should be mapped onto available computational resources. The efficiency gains achievable in practice therefore depend not only on partitioning quality, but also on the scheduling strategy used to assign subsystem tasks to processing cores.

In this context, the dependency DAG produced by structural analysis becomes the direct input to the parallel scheduling layer. The scheduling stage is responsible for ordering subsystem execution, balancing workload across cores, and minimizing makespan under precedence constraints. The next section formalizes this scheduling problem and presents the ten algorithms evaluated in this study.

4 Parallel scheduling framework

Following structural decomposition, the resulting subsystems and their dependencies form a Directed Acyclic Graph (DAG). The second layer of acceleration is therefore achieved through parallel scheduling: dependent subsystems are solved sequentially, while independent subsystems are processed concurrently. This section formalizes the scheduling problem, presents the computational cost model, and describes the evaluated scheduling strategies.

4.1 DAG-based scheduling model

The resolution process is modeled as a weighted DAG $G = (T, E, W)$, where:

- $T = \{T_1, T_2, \ldots, T_n\}$ is the set of tasks, each corresponding to the numerical resolution of a subsystem,
- E is the set of directed edges representing precedence constraints,
- $W = \{w_1, w_2, \ldots, w_n\}$ denotes the computational cost of each task.

An edge $e_{i,j} \in E$ indicates that task T_j can begin only after the completion of T_i. The objective is to map tasks onto P identical processors such that the overall makespan $C_{\max}$—defined as the completion time of the final task—is minimized.

4.2 Computational cost estimation

Task execution time is estimated using the cubic complexity associated with Jacobian-based iterative solvers [8, 17]. For a subsystem with n_i variables, the estimated execution time is:

$$ET(T_i) = C \cdot n_i^3 \tag{1}$$

The constant C was empirically calibrated on an AMD Ryzen 7 4750U using `scipy.optimize.fsolve`. The estimation was performed using our open-source benchmarking code, available on GitHub [43]. Benchmarking across a wide range of system sizes and multiple equation types—including polynomial, trigonometric, and transcendental forms—yielded the empirically determined values shown in Table 2. Since our experimental framework relies specifically on polynomial-based operations, we adopt the corresponding value of $C = 0.99985 \times 10^{-9}$ seconds as a conservative baseline for all subsequent simulations.

Table 2: Empirical computational constants C

Equation Type	C (seconds)
Simple Polynomial	0.99985×10^{-9}
Trigonometric	0.96747×10^{-9}
Exponential	0.94805×10^{-9}
Cubic Polynomial	0.96614×10^{-9}
Mixed Transcendental	0.95458×10^{-9}

4.3 Scheduling strategy categories

To explore trade-offs between scheduling overhead and execution efficiency, ten algorithms are implemented and grouped into three categories [40, 37].

4.3.1 Critical path–aware schedulers

These strategies prioritize tasks that lie on or near the longest dependency chain of the DAG [40].

- **CP_Priority:** Explicitly identifies the critical path and assigns highest priority to its tasks [8].
- **HEFT:** Uses upward rank (bottom-level) to prioritize tasks with longer remaining execution chains and assigns each task to the processor minimizing finish time [38].
- **ETF:** Selects the task–processor pair yielding the earliest possible start time using top-level and bottom-level metrics [39].
- **CPOP:** Computes combined upward and downward ranks to identify the complete critical path and assigns its tasks to a single processor [38].

4.3.2 Heuristic-based schedulers

These strategies rely on structural graph properties rather than explicit critical path computation [37].

- **Level-by-Level:** Executes tasks by topological depth [40].
- **Largest First:** Prioritizes tasks with the largest execution time [37].
- **Smallest First:** Prioritizes shortest tasks to rapidly unlock successors [37].
- **Most Successors First:** Prioritizes tasks with highest out-degree [40].

4.3.3 Baseline schedulers

- **FIFO:** Schedules tasks in order of readiness [37].
- **Random:** Randomly selects from ready tasks [37].

4.4 Parallel execution architecture

The scheduling framework is implemented using a centralized dispatcher. A Task Manager maintains a dynamic ready queue containing tasks whose dependencies have been satisfied. When a processor becomes available, a scheduling policy selects the next task from this queue.

Upon task completion, successor dependencies are updated and newly ready tasks are inserted into the queue. Lightweight synchronization primitives are used to minimize overhead, ensuring that scheduling costs do not offset parallelization gains.

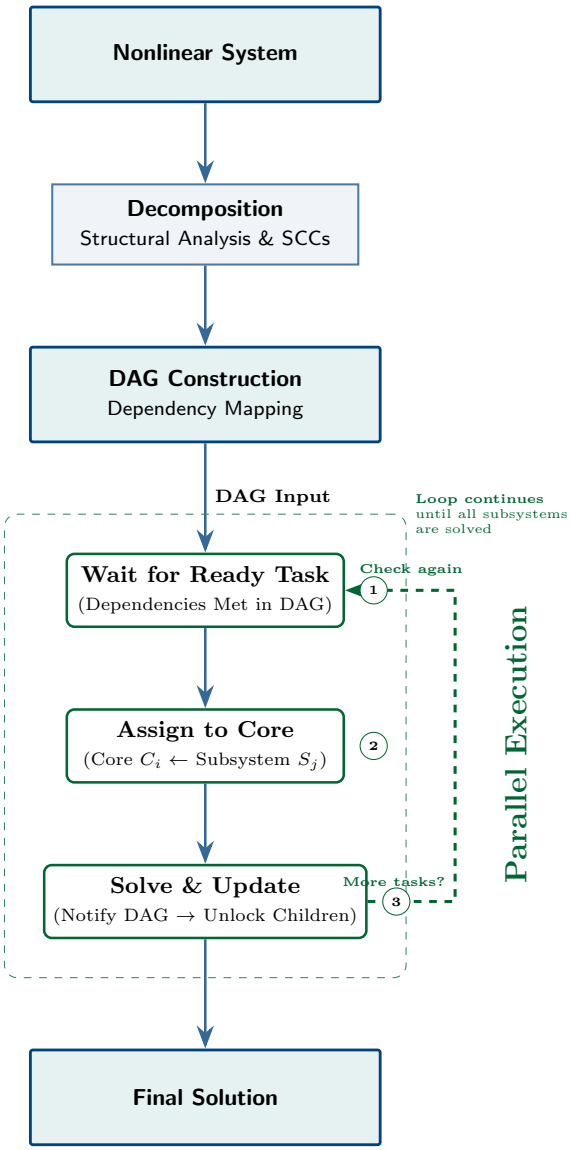

Figure 1: The two-phase acceleration pipeline: Decomposition and parallelisation.

Figure 1 illustrates the complete pipeline from decomposition to parallel execution.

4.5 Scope and experimental abstraction

Although nonlinear system solving involves decomposition, dependency graph construction, and parallel execution, this study focuses specifically on the scheduling stage. Instead of reimplementing decomposition, we generate synthetic DAGs that emulate realistic subsystem dependencies. This enables systematic control of graph size, depth, and density while isolating scheduling effects from decomposition-specific behavior.

Each node is mapped to a nonlinear subsystem solved using `fsolve`. The scheduling algorithms operate on these

DAGs exactly as they would on graphs produced by actual decomposition.

4.6 Illustrative example: critical path–driven scheduling analysis

To analytically demonstrate the role of critical path prioritization, we consider the dependency graph depicted in Figure 2. The graph consists of six subsystems A, B, C, D, E, and F, where directed edges represent strict precedence constraints. Each node corresponds to a nonlinear subsystem whose computational cost is estimated using the cubic model $ET(T_i) = C \cdot n_i^3$. For clarity of exposition, we set $C = 1$.

The subsystem sizes are $n = \{2, 3, 4, 3, 2, 2\}$ for (A, B, C, D, E, F), yielding execution times:

$$ET(A) = 8, \quad ET(B) = 27, \quad ET(C) = 64,$$
$$ET(D) = 27, \quad ET(E) = 8, \quad ET(F) = 8.$$

Critical path identification The weighted longest path in the DAG is:

$$A \to C \to E,$$

with cumulative cost:

$$8 + 64 + 8 = 80.$$

This path constitutes the *critical path*, and therefore establishes a theoretical lower bound on the makespan:

$$C_{\max} \geq 80.$$

No valid schedule, regardless of processor count, can complete the system in less than 80 time units due to these structural dependencies.

Scheduling with two processors Assume a homogeneous platform with $P = 2$ processors. At $t = 0$, only subsystem A is executable. After its completion at $t = 8$, subsystems B, C, and D become simultaneously ready. Since only two processors are available, the scheduler must select two tasks while postponing one.

Three principal decision patterns arise:

1. **Critical-path prioritization** Schedule C together with either B or D. In both cases, execution along the critical path proceeds without interruption. The resulting makespan equals the lower bound:

$$C_{\max} = 80.$$

2. **Critical-path delay** Schedule B and D, postponing C. This decision interrupts progression along the longest dependency chain. Consequently, subsystem E is also delayed, producing:

$$C_{\max} = 103.$$

The 23-unit performance degradation corresponds exactly to the idle propagation effect induced by delaying the dominant weighted path. Although all three scheduling choices satisfy precedence constraints, only the critical-path-aware strategies achieve the structural lower bound.

Structural interpretation This example highlights a fundamental property of DAG scheduling: under resource constraints, local scheduling decisions must account for global dependency structure. The makespan is governed not by aggregate workload alone, but by the interaction between processor availability and the longest weighted dependency chain.

Delaying non-critical tasks may increase processor utilization momentarily, yet postponing a critical task irreversibly shifts the earliest possible completion time of all its successors. Hence, optimal or near-optimal scheduling requires explicit consideration of bottom-level (longest-path) metrics.

Implications for scheduler design The example formally justifies the integration of critical-path metrics in algorithms such as CP_Priority, HEFT, ETF, and CPOP. These schedulers estimate upward and/or downward ranks to approximate the longest weighted paths and bias allocation decisions toward tasks with high structural impact.

Conversely, heuristics that ignore global path structure (e.g., FIFO or purely size-based selection) may inadvertently delay critical tasks and thus inflate the makespan, particularly under limited processor counts.

Figure 2 therefore serves not merely as an illustrative diagram, but as a structural proof-of-concept demonstrating how dependency topology and weighted path dominance dictate achievable performance bounds in parallel nonlinear system resolution.

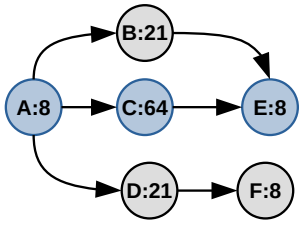

Figure 2: Example dependency graph illustrating critical path scheduling.

4.7 Illustrative critical path scheduling example

To demonstrate the operation of our parallel scheduling framework, we consider a representative nonlinear system of 31 equations with 31 unknowns. Each equation defines a numerical subsystem, forming the set of tasks to be scheduled. The algebraic formulation is presented in Figure 3,

with all variables labeled x_i or y_i and known constants substituted where applicable. While the full system is detailed, the focus of scheduling analysis is on the dependency structure among subsystems, captured by a Directed Acyclic Graph (DAG).

$$\begin{cases} (x_a - x_b)^2 + (y_a - y_b)^2 = 1, & (x_h - x_i)^2 + (y_h - y_i)^2 = 2, \\ (x_b - x_c)^2 + (y_b - y_c)^2 = 1, & (x_h - x_j)^2 + (y_h - y_j)^2 = 1, \\ (x_a - x_c)^2 + (y_a - y_c)^2 = 2, & (x_i - x_j)^2 + (y_i - y_j)^2 = 1, \\ (x_a - x_f)^2 + (y_a - y_f)^2 = 1, & (x_j - x_k)^2 + (y_j - y_k)^2 = 2, \\ (x_c - x_f)^2 + (y_c - y_f)^2 = 1, & (x_i - x_k)^2 + (y_i - y_k)^2 = 1, \\ (x_a - x_d)^2 + (y_a - y_d)^2 = 8, & (x_j - x_l)^2 + (y_j - y_l)^2 = 1, \\ (x_c - x_e)^2 + (y_c - y_e)^2 = 8, & (x_k - x_l)^2 + (y_k - y_l)^2 = 1, \\ (x_p - x_d)^2 + (y_p - y_d)^2 = 1, & (x_a - x_m)^2 + (y_a - y_m)^2 = 1, \\ (x_p - x_e)^2 + (y_p - y_e)^2 = 1, & (x_b - x_m)^2 + (y_b - y_m)^2 = 2, \\ (x_p - x_f)^2 + (y_p - y_f)^2 = 2, & (x_a - x_o)^2 + (y_a - y_o)^2 = 13, \\ (x_d - x_e)^2 + (y_d - y_e)^2 = 2, & (x_b - x_n)^2 + (y_b - y_n)^2 = 13, \\ (x_b - x_g)^2 + (y_b - y_g)^2 = 2, & (x_q - x_o)^2 + (y_q - y_o)^2 = 2, \\ (x_c - x_g)^2 + (y_c - y_g)^2 = 1, & (x_q - x_n)^2 + (y_q - y_n)^2 = 1, \\ (x_b - x_h)^2 + (y_b - y_h)^2 = 1, & (x_q - x_m)^2 + (y_q - y_m)^2 = 2, \\ (x_g - x_h)^2 + (y_g - y_h)^2 = 1, & (x_o - x_n)^2 + (y_o - y_n)^2 = 1, \\ (x_g - x_i)^2 + (y_g - y_i)^2 = 1 \end{cases}$$

Figure 3: Representative nonlinear system of 31 equations.

Dependency Graph and Critical Path. The computational dependencies between subsystems are abstracted as a DAG, where each node corresponds to a subsystem derived from an equation or a small cluster of equations. Directed edges encode precedence constraints: an edge $T_i \to T_j$ indicates that T_j cannot commence until T_i has been resolved. Figure 4 illustrates this DAG with nodes labeled according to their subsystem index. The critical path, determined using the estimated execution times of each subsystem (modeled as $ET(T_i) = C \cdot n_i^3$), is highlighted in red to emphasize the minimum achievable schedule length.

Illustrative Scheduling Analysis. Consider scheduling this system on a machine with two available cores. Tasks on the critical path are prioritized to minimize overall completion time. Let the estimated resolution times for the critical path subsystems be calculated using the cubic complexity model (Equation (1)).

Scheduling proceeds iteratively as follows:

1. Initialize the *Ready Queue* with tasks that have no incoming edges.
2. Assign ready tasks to available cores.
3. Upon task completion, update dependent tasks' states. Newly ready tasks are added to the queue.
4. Repeat until all tasks are completed.

Three illustrative scheduling scenarios underscore the importance of critical path prioritization:

– **Scenario 1:** Tasks on the critical path scheduled first → Minimum total completion time.

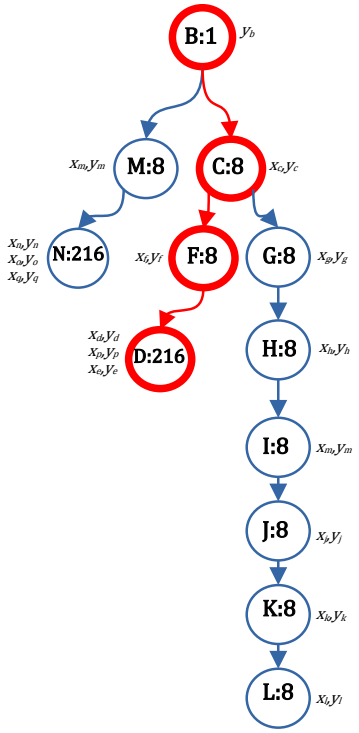

Figure 4: Conceptual DAG of the nonlinear system. The critical path is highlighted in red.

– **Scenario 2:** Non-critical tasks prioritized → Increased total completion time due to idle cores along the critical path.

– **Scenario 3:** Random scheduling → Maximum total completion time, demonstrating the effectiveness of heuristics and critical path awareness.

This example concretely demonstrates that critical path-based scheduling significantly reduces makespan and validates the use of both analytical execution time modeling and DAG-based dependency representation in the evaluation of parallel nonlinear solvers.

5 Performance evaluation: experimental results and analysis

This section provides a comprehensive evaluation of ten DAG scheduling algorithms applied to synthetic nonlinear systems containing up to 50,000 variables. The analysis investigates how scheduler selection, dependency density, and core allocation collectively influence parallel performance, scalability, and scheduling overhead.

5.1 Experimental setup

The objectives of this experimental study are threefold: (i) to quantify parallel speedup as the core count scales from 2 to 8 cores, (ii) to compare scheduler behavior across structurally distinct DAG regimes (sparse, balanced, dense), and

(iii) to identify scheduler–core configurations that maximize efficiency for each workload class.

Experiments were conducted on an AMD Ryzen 7 4750U processor (8 physical cores, 16 threads) equipped with 16 GB DDR4 RAM. The system was running a 64-bit Linux operating system. The implementation was developed in `Python 3.11.10`, utilizing the `NumPy` [47] and `SciPy` [44] libraries for numerical computation, and `NetworkX` [48] for graph construction. Parallel execution was implemented using Python's multiprocessing module to exploit multicore capabilities. All experiments were executed under identical system and software conditions to ensure consistency and reproducibility.

Synthetic DAGs were generated using a probabilistic edge construction mechanism. For any ordered node pair (i, j) with $i < j$, a directed edge $i \to j$ was created with probability ρ, ensuring acyclicity while allowing precise control over dependency density.

Three distinct structural regimes were evaluated:

- **Sparse ($\rho = 0.15$):** Minimal dependencies with high task-level parallelism.
- **Balanced ($\rho = 0.30$):** Moderate connectivity combining parallel and serial segments.
- **Dense ($\rho = 0.50$):** Strong dependency constraints limiting concurrent execution.

These ρ values were selected to produce clearly separable parallelism regimes without structural overlap.

Ten schedulers were evaluated using 2, 4, and 8 cores, covering the range of cores available on the target processor while capturing low, moderate, and high concurrency regimes. Sequential execution served as the baseline for speedup computation. Problem sizes ranged from 6,000 to 50,000 variables, selected to ensure that computational workloads are sufficiently large to amortize scheduling overhead while remaining representative of practical engineering systems. Each configuration was repeated three times ($n = 3$), and mean values with standard deviations are reported. Although the number of trials is limited, consistent ranking patterns and substantial performance differences support the reliability of observed trends.

5.2 Scheduler performance analysis

Table 3 reveals pronounced variation across structural regimes and core counts. Sparse graphs achieve maximum speedup of $1.53\times$ at 2 cores using HEFT. At 4 cores, multiple schedulers converge near $1.55\times$ speedup, with Level-by-Level, Most Successors First, and CPOP achieving $1.56\times$. However, scaling to 8 cores yields diminishing returns (1.51–$1.54\times$), indicating structural saturation where dependency constraints limit further parallelization gains.

Balanced graphs exhibit more stable performance across schedulers, with Level-by-Level achieving $1.21\times$ at 2 cores and Most Successors First sustaining competitive performance ($1.08\times$) at 4 cores. Dense graphs demonstrate stronger sensitivity to scheduling decisions at low core counts, where HEFT provides $1.13\times$ acceleration; however, performance degrades at higher core counts (down to 0.75–$0.84\times$ at 8 cores), indicating overhead exceeds parallelization benefits.

Overall, no scheduler dominates across all regimes, underscoring the context-dependent nature of scheduling effectiveness. Notably, 4 cores appears optimal for sparse graphs, while balanced and dense graphs show limited or negative scaling beyond 2 cores.

Figure 5 illustrates convergence behavior for the representative sparse instance ($N = 24{,}000$). At 2 cores, the makespan ranges from 48.89 s (HEFT) to 58.27 s (Smallest First), yielding a spread of 9.38 s (19.2% variance). At 4 cores, the range narrows substantially to 1.61 s (47.95–49.56 s, 3.4% variance), and at 8 cores the difference further reduces to only 0.97 s (48.82–49.79 s, 2.0% variance).

Although this figure presents only the sparse case at a single problem size for clarity, similar convergence patterns were observed across other problem sizes and dependency densities (see Table 3). These results indicate that as core count increases, structural constraints increasingly limit scheduler differentiation, reducing the relative impact of prioritization heuristics.

Three trends emerge:

1. Optimal scheduler varies with core count and graph structure.
2. Scheduler differentiation decreases dramatically as core count increases (19.2% at 2 cores $\to$ 2.0% at 8 cores for sparse graphs).
3. Simple schedulers remain competitive when parallelism is structurally constrained, with several achieving near-optimal performance at 4+ cores.

This progression reflects a transition from scheduler-sensitive performance at low core counts to dependency-limited performance at higher counts, where the critical path and structural dependencies become the dominant performance bottleneck.

5.2.1 Parallel speedup contributions

Table 4 separates total acceleration into two layers: (i) structural decomposition and (ii) parallel execution.

Structural decomposition is the dominant contributor. Moving from the monolithic solver to the decomposed single-core version yields $184.6\times$ (sparse), $74.9\times$ (balanced), and $65.1\times$ (dense) speedups, demonstrating that factorization of dependencies is the primary performance driver.

Parallelization provides an additional multiplicative gain whose magnitude depends on graph structure.

Table 3: Scheduler performance matrix showing speedup relative to sequential execution on AMD Ryzen 7 4750U (8 cores 16 GB RAM) for sparse, balanced, and dense graphs. Bold values indicate the best per column.

Scheduler	2 Cores			4 Cores			8 Cores		
	S	M	D	S	M	D	S	M	D
cp_priority	1.45	1.08	1.10	**1.55**	0.94	0.88	1.52	0.86	0.80
heft	**1.53**	1.16	**1.13**	**1.55**	1.01	**0.94**	**1.53**	0.86	0.79
etf	1.50	1.09	1.09	1.51	1.02	0.93	1.52	0.82	0.75
level_by_level	1.45	**1.21**	1.07	**1.56**	1.07	0.93	1.51	0.88	0.82
largest_first	1.45	1.16	1.04	1.53	1.07	0.92	1.51	0.88	0.81
smallest_first	1.29	1.19	0.95	1.54	1.07	0.91	**1.54**	0.88	0.81
most_succ	1.47	1.20	0.96	**1.56**	**1.08**	0.92	1.51	0.89	0.81
cpop	1.42	0.98	0.95	**1.56**	1.08	0.93	1.52	0.88	0.82
fifo	1.48	1.00	1.00	**1.55**	1.03	0.92	1.52	0.87	**0.84**
random	1.47	0.99	0.97	**1.55**	0.99	0.93	**1.53**	0.86	0.80

S = Sparse, M = Balanced, D = Dense. Bold indicates best per column.

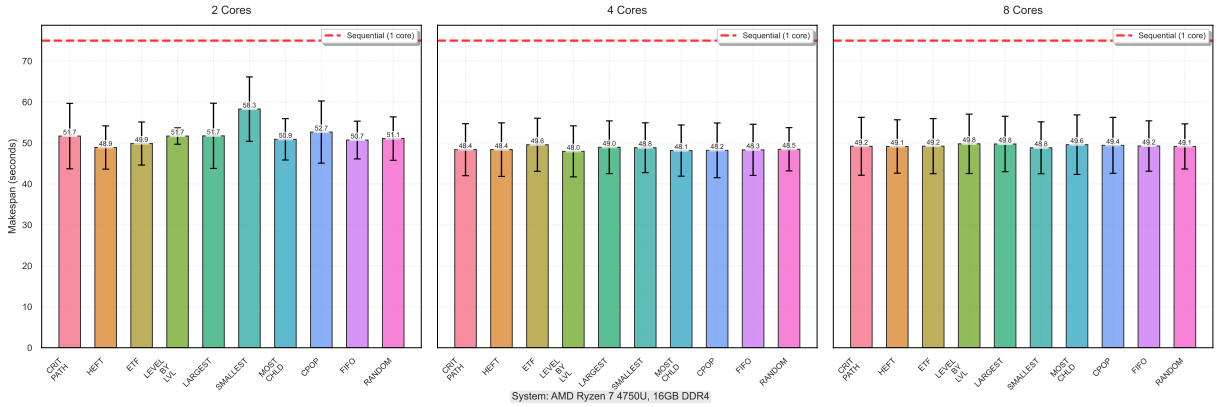

Figure 5: Makespan comparison across schedulers for 2, 4, and 8 cores. Platform: AMD Ryzen 7 4750U (8 cores), 16 GB DDR4 RAM. Problem configuration: Sparse graph, $N = 24{,}000$ variables. Runtime measured in seconds. Error bars show standard deviation ($n = 3$).

For sparse graphs, parallel gains remain stable across core counts ($1.53\times$–$1.56\times$), reaching a peak total acceleration of $288.7\times$ at 4 cores. This stability indicates sufficient concurrency and limited scheduling overhead.

Balanced graphs show moderate scalability: the additional gain peaks at $1.21\times$ (2 cores) and decreases as core count increases, reflecting growing dependency constraints.

Dense graphs exhibit strong coupling effects. While 2 cores yield a $1.13\times$ gain, performance degrades at higher core counts ($0.94\times$ at 4 cores, $0.84\times$ at 8 cores), indicating that synchronization and scheduling overhead exceed exploitable parallelism.

Overall, the results confirm that decomposition provides the fundamental acceleration, while parallel scheduling amplifies performance only when structural independence is sufficient. Scalability is therefore bounded by dependency density and critical-path dominance.

5.2.2 Overhead-driven performance tradeoffs

Figure 6 presents a representative large-scale instance ($N = 24{,}000$). Across all tested large problem sizes (6,000–50,000 variables), the same qualitative behavior was observed.

Overhead differences are visible at 2 cores but progressively converge at 4 and 8 cores. The overhead percentage remains relatively stable (typically 10–15%) across core counts and system sizes, indicating that scheduling cost scales proportionally with workload.

The observed reduction in speedup at higher core counts is therefore driven primarily by structural constraints (dependency density and critical-path dominance), rather than increasing coordination overhead. This trend is consistent across sparse, balanced, and dense regimes.

5.2.3 Scalability constraints: cache and memory bandwidth analysis

The experimental results indicate that increasing the core count from 4 to 8 does not produce proportional speedup, and in certain cases leads to marginal performance degradation. This behavior is primarily attributed to cache hierarchy limitations and shared memory bandwidth contention, which are inherent characteristics of contemporary laptop-class multicore processors.

Table 4: Acceleration breakdown from structural decomposition and parallel execution for sparse, balanced, and dense graphs. Peak total speedups are highlighted in bold.

Structure	Layer 1	Layer 2: Additional Parallel Gain			Peak Total
	Decomposition (1C)	2C	4C	8C	Acceleration
Sparse	184.6×	1.53×	**1.56**×	1.54×	**288.7**× **(4C)**
Balanced	74.9×	**1.21**×	1.08×	0.89×	**90.6**× **(2C)**
Dense	65.1×	**1.13**×	0.94×	0.84×	**73.4**× **(2C)**

Layer 1 = speedup from structural decomposition only. Layer 2 = multiplicative gain from parallel execution.

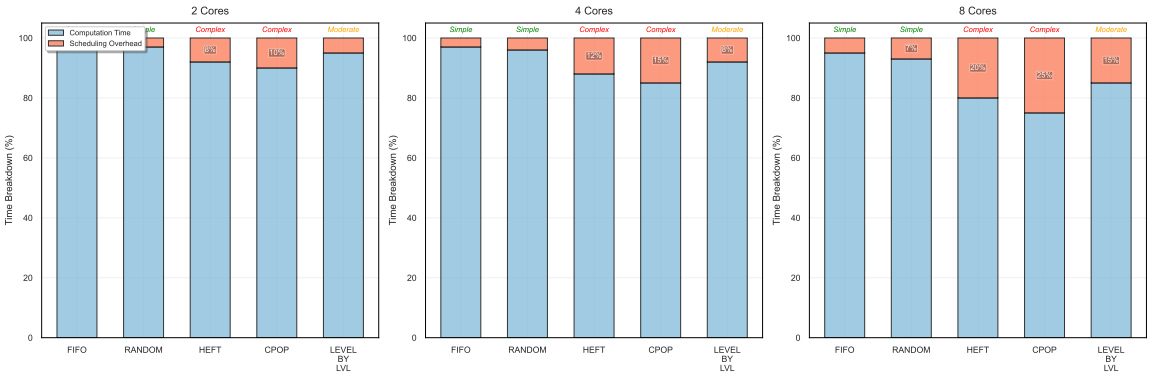

Figure 6: Scheduling overhead as a percentage of total execution time for the sparse graph configuration ($N = 24{,}000$). Results are shown for 2, 4, and 8 cores on an AMD Ryzen 7 4750U (8 cores, 16 threads) with 16 GB DDR4 RAM. Overhead percentages are averaged over $n = 3$ independent trials.

Cache Capacity and Working-Set Pressure. Modern mobile processors provide per-core L2 cache capacities typically ranging from 512 KB to 1.25 MB, with a shared last-level cache (L3) of limited size. In the present study, each core processes matrix blocks ranging from 1000×1000 to several thousands per dimension. For double-precision arithmetic (8 bytes per element), a single 1000×1000 matrix requires approximately 8 MB of storage, which substantially exceeds the private L2 capacity and often approaches or exceeds the effective per-core share of the L3 cache.

As parallel workers increase, the aggregate working set scales proportionally, while cache capacity remains fixed. This results in elevated cache miss rates and increased traffic to the shared last-level cache and main memory. The additional latency incurred by frequent cache evictions and refills reduces the effective computational throughput of each core, limiting parallel efficiency.

Memory Bandwidth Saturation. Laptop processors typically rely on dual-channel DDR4 or DDR5 memory subsystems, offering finite peak bandwidth. Dense numerical workloads, particularly matrix-based computations, exhibit substantial memory traffic due to repeated loading of operands and writing of intermediate results.

When a limited number of cores are active, the available memory bandwidth per core is sufficient to sustain near-optimal execution. However, as concurrency increases, the bandwidth available to each core decreases, eventually transitioning the workload into a memory-bound regime. Under such conditions, cores stall while waiting for data transfers, and additional parallel workers primarily increase contention rather than useful computation.

The observed performance plateau beyond four cores is therefore consistent with well-established memory-wall phenomena and multicore bandwidth-sharing effects, as described in the Roofline performance model and contemporary memory-system analyses [45, 46], rather than deficiencies in the proposed scheduling framework.

These scalability characteristics are typical of laptop-class shared-memory architectures. Systems equipped with larger last-level caches and multi-channel memory subsystems (e.g., quad-channel server platforms) would be expected to demonstrate improved scaling behavior for equivalent workloads.

6 Conclusion

This study systematically evaluated DAG scheduling strategies for the parallel solution of decomposed nonlinear equation systems. The results demonstrate that performance is governed by the interaction between dependency structure, scheduling overhead, and core allocation.

Structural decomposition provides the dominant acceleration, while parallel execution acts as a multiplicative enhancement layer. Sparse graphs benefit most from parallelism, achieving speedups up to $1.55\times$ at 4 cores with stable scalability. In contrast, balanced and dense graphs exhibit limited scalability beyond 2 cores due to increased dependency constraints and overhead effects.

Scheduler differentiation is significant at low core counts

but diminishes as parallelism increases, with makespan variance decreasing from 19.2% at 2 cores to 2.0% at 8 cores for sparse graphs. This convergence indicates that structural critical paths ultimately bound performance, regardless of scheduling sophistication.

Parallel execution amplifies decomposition gains by 1.13–1.53× at 2 cores, producing total accelerations exceeding 280× relative to monolithic solving for sparse structures. However, the multiplicative benefit decreases with graph density, reflecting the limiting effect of structural coupling on available concurrency. Furthermore, the achievable speedup is inherently bounded by cache hierarchy constraints and shared memory bandwidth limitations, consistent with well-established memory-wall phenomena [45, 46].

Overall, the findings demonstrate that effective deployment requires structure-aware core allocation and workload-adaptive scheduling rather than maximal hardware utilization. Future work will investigate adaptive scheduling mechanisms that dynamically adjust prioritization strategies based on runtime dependency characteristics, memory-aware scheduling approaches that explicitly account for cache hierarchy and bandwidth constraints, and evaluation across heterogeneous platforms, including multi-channel server architectures where improved scaling behavior is anticipated.

Acknowledgement

AI-assisted tools were used to support drafting and language refinement of the manuscript. All scientific ideas, methods, experiments, and conclusions are the authors' own, and the authors are fully responsible for the content. This research received no external funding.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, McGraw-Hill, New York, 1988.

[2] S. H. Strogatz, *Nonlinear Dynamics and Chaos with Student Solutions Manual: With Applications to Physics, Biology, Chemistry, and Engineering*, CRC Press, 2018. DOI: 10.1201/9780429492563

[3] J. D. Anderson Jr, *Fundamentals of Aerodynamics*, McGraw-Hill, New York, 1991.

[4] G. Payette and J. Reddy, "A nonlinear finite element framework for viscoelastic beams based on the high-order Reddy beam theory," *J. Eng. Mater. Technol.*, vol. 135, no. 1, p. 011005, 2013. DOI: 10.1115/1.4007562

[5] M. C. Boyce and E. M. Arruda, "Constitutive models of rubber elasticity: a review," *Rubber Chem. Technol.*, vol. 73, no. 3, pp. 504–523, 2000. DOI: 10.5254/1.3547602

[6] L. M. P. Ghilardi, S. Naik, E. Martelli, F. Casella, and L. T. Biegler, "Economic Nonlinear Model Predictive Control for cyclic gas pipeline operation," *Comput. Chem. Eng.*, p. 109039, 2025. DOI: 10.1016/j.compchemeng.2025.109039

[7] H. K. Khalil and J. W. Grizzle, *Nonlinear Systems*, vol. 3, Prentice Hall, Upper Saddle River, NJ, 2002.

[8] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, SIAM, 1995. DOI: 10.1137/1.9781611970944

[9] R. Burden and J. Faires, *Numerical Analysis*, Brooks/Cole, Cengage Learning, Boston, USA, 2011.

[10] W. H. Press, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, Cambridge University Press, 2007.

[11] C. G. Broyden, "A class of methods for solving nonlinear simultaneous equations," *Math. Comput.*, vol. 19, no. 92, pp. 577–593, 1965. DOI: 10.2307/2003941

[12] J. E. Dennis Jr and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, SIAM, 1996. DOI: 10.1137/1.9781611971200

[13] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *J. Soc. Ind. Appl. Math.*, vol. 11, no. 2, pp. 431–441, 1963. DOI: 10.1137/0111030

[14] A. Morgan, *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems*, SIAM, 2009. DOI: 10.1137/1.9780898719031

[15] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, 1999. DOI: 10.1007/b98874

[16] S. P. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004. DOI: 10.1017/CBO9780511804441

[17] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013. DOI: 10.56021/9781421407944

[18] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, SIAM, 2000. DOI: 10.1137/1.9780898719468

[19] S. Zhao, X. Dai, and I. Bate, "DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4019–4038, Dec. 2022, DOI: 10.1109/TPDS.2022.3177046.

[20] L. Bendiaf, A. Harbouche, M. A. Tahraoui, and F. Z. Lebbah,"An innovative task scheduling method utilizing the knapsack algorithm in heterogeneous computing systems," *Informatica*, vol. 48, pp. 89–104, 2024. DOI: 10.31449/inf.v48i16.5765

[21] S. Chang, X. Zhao, Z. Liu, and Q. Deng, "Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores," *Journal of Systems Architecture*, vol. 105, p. 101704, Apr. 2020, DOI: 10.1016/j.sysarc.2019.101704.

[22] X. Feng, C. Shushan, H. Xingxing, H. Shujuan, and Z. Wenjuan, "A new direct acyclic graph task scheduling method for heterogeneous Multi-Core processors," *Computers and Electrical Engineering*, vol. 104, p. 108464, Dec. 2022, DOI: 10.1016/j.compeleceng.2022.108464.

[23] P. D. Michailidis and K. G. Margaritis, "Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP," *Journal of Computational and Applied Mathematics*, vol. 236, no. 3, pp. 326–341, Nov. 2011, DOI: 10.1016/j.cam.2011.07.002.

[24] X. Xiao, "ILS-YOLO: An improved list scheduling algorithm for YoloLite DAGs in vehicular edge computing," *Informatica*, vol. 49, pp. 237–250, 2025. DOI: 10.31449/inf.v49i29.11851

[25] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 1, pp. 15–44, Jan. 2010, doi: 10.1002/cpe.1467.

[26] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 3, pp. 305–321, Mar. 2012, doi: 10.1002/cpe.1829.

[27] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, pp. 1–11, doi: 10.1145/1654059.1654079.

[28] D. Lukarski, "Parallel sparse linear algebra for multicore and many-core platforms: Parallel solvers and preconditioners," Ph.D. dissertation, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2012.

[29] P. D'Ambra, F. Durastante, and S. Filippone, "Parallel Sparse Computation Toolkit," *Software Impacts*, vol. 15, p. 100463, 2023, DOI: 10.1016/j.simpa.2022.100463.

[30] M. Belmabrouk and M. Marrakchi, "Design, analysis and performance evaluation of parallel algorithms for solving triangular linear systems on multicore platforms," *RAIRO - Operations Research*, vol. 55, no. 2, pp. 545–559, Mar. 2021, DOI: 10.1051/ro/2019114.

[31] S. Marrakchi and M. Jemni, "Static scheduling with load balancing for solving triangular band linear systems on multicore processors," *Fundamenta Informaticae*, vol. 179, no. 1, pp. 35–58, Jan. 2021, DOI: 10.3233/FI-2021-2012.

[32] L. Mochurad and N. Boyko, "Solving systems of nonlinear equations on multi-core processors," in *Proceedings of the Conference on Computer Science and Information Technologies*, 2019, pp. 90–106, DOI: 10.1007/978-3-030-33695-0_8.

[33] V. Tavakkoli, K. Mohsenzadegan, J. C. Chedjou, and K. Kyamakya, "Contribution to speeding-up the solving of nonlinear ordinary differential equations on parallel/multi-core platforms for sensing systems," *Sensors*, vol. 20, no. 21, p. 6130, Oct. 2020, DOI: 10.3390/s20216130.

[34] F. González, A. Luaces, U. Lugrís, and M. González, "Non-intrusive parallelization of multibody system dynamic simulations," *Computational Mechanics*, vol. 44, no. 4, pp. 493–504, Oct. 2009, doi: 10.1007/s00466-009-0386-3.

[35] S. Ait-Aoudia, R. Jegou, and D. Michelucci, "Reduction of constraint systems," in *Proceedings of the International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS'93)*, Alvor, Portugal, 1993, pp. 331–340.

[36] A. L. Dulmage and N. S. Mendelsohn, "Coverings of bipartite graphs," *Can. J. Math.*, vol. 10, pp. 517–534, 1958. DOI: 10.4153/CJM-1958-052-0

[37] O. Sinnen, "Task Scheduling for Parallel Systems," *John Wiley & Sons*, 2007. DOI: 10.1002/9780470170575

[38] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002. DOI: 10.1109/71.993206

[39] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times,"*SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, 1989. DOI: 10.1137/0218016

[40] Kwok, Y. K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471. DOI 10.1145/344588.344618

[41] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004. DOI: 10.1109/TPDS.2004.1264795

[42] L.-C. Canon and E. Jeannot. Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010. DOI: 10.1109/TPDS.2009.84

[43] A. Moussaoui, F. Belhadj, E. Y. Gueddoudj, S. Bouamama and N. Boukhari. ParallelEqSolver: Parallel nonlinear equations solver with dependency-aware scheduling. *GitHub repository*, 2025. https://github.com/adelmoussaoui/paralleleqsolver

[44] P. Virtanen et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[45] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: 10.1145/1498765.1498785

[46] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proceedings of the 5th IEEE International Memory Workshop (IMW)*, 2013, pp. 21–25. DOI: 10.1109/IMW.2013.6582088

[47] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020. DOI: 10.1038/s41586-020-2649-2

[48] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, Pasadena, CA, 2008, pp. 11–15. DOI: 10.25080/TCWV9851