UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ALISSON DOS PASSOS FUMACO

Visualizing Algorithmic Progression in Task-Based Parallelism: A Visual Analytics Methodology for Evaluating Dynamic Schedulers

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
September 2026

CIP — CATALOGING-IN-PUBLICATION

dos Passos Fumaco, Alisson

Visualizing Algorithmic Progression in Task-Based Parallelism: A Visual Analytics Methodology for Evaluating Dynamic Schedulers / Alisson dos Passos Fumaco. – Porto Alegre: PPGC da UFRGS, 2026.

69 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2026. Advisor: Lucas Mello Schnorr.

1. Task-Based Parallelism. 2. Performance Visualization. 3. Visual Analytics. 4. Dynamic Scheduling. 5. High-Performance Computing. 6. Tiled QR Factorization. 7. Dense Linear Algebra. 8. StarPU. 9. StarVZ. 10. Execution Traces. I. Mello Schnorr, Lucas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Marcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Pós-Graduação: Prof[a]. Claudia Wasserman
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do PPGC: Prof. Gabriel Luca Nazar
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

A todos que processam o amanhã, transformando cada bit de esforço no degrau que conduzirá o Brasil ao seu lugar de direito na história.

AGRADECIMENTOS

Primeiramente, agradeço ao meu orientador, Prof. Dr. Lucas Schnorr, pela oportunidade, pelas discussões enriquecedoras e, principalmente, pela enorme paciência em lidar com o meu estilo um pouco rebelde e, por vezes, resistente a críticas. Sua orientação precisa e visão crítica foram fundamentais para que este trabalho chegasse ao fim.

Agradeço aos colegas do Laboratório de Processamento Paralelo e Distribuído (LPPD) não apenas pelas trocas de ideias e parceria no dia a dia, mas também pelas indispensáveis partidas de sinuca e pelas piadas duvidosas que ajudaram a manter a sanidade do grupo. Um agradecimento especial à equipe técnica do PCAD por manter a infraestrutura operante.

Um agradecimento muito especial a duas pessoas essenciais na minha trajetória e que escreveram minhas cartas de recomendação: à Daniela Bagatini, minha professora de introdução à programação e orientadora do meu primeiro projeto de extensão, que abriu as portas da computação para mim; e à Gabriela Bragagnolo, minha antiga chefe, por ter confiado no meu trabalho e me dado a oportunidade de assumir maiores responsabilidades. Aproveito também para agradecer ao meu atual chefe, Alexandre Farias, por toda a compreensão e flexibilidade com os meus horários complicados durante este período.

À UFRGS e ao Instituto de Informática (INF), por fornecerem um ambiente de ensino público de excelência.

Agradeço profundamente à minha família e aos meus amigos por todo o apoio, pela compreensão durante as incontáveis horas em que estive ausente programando ou escrevendo, e por fazerem toda essa jornada valer a pena.

Por fim, meu profundo agradecimento a todas as pessoas que acreditam em um Brasil grande. Um agradecimento especial aos pesquisadores e profissionais que tiveram a oportunidade de estudar no exterior, mas escolheram retornar para construir e desenvolver o conhecimento científico no nosso país. E, acima de tudo, aos alunos que virão com a ousadia de buscar o verdadeiro conhecimento, desafiando um sistema que se conformou em ser apenas uma fábrica de diplomas.

ABSTRACT

The shift towards multicore architectures has increasingly positioned task-based parallelism as a prominent paradigm for high-performance computing (HPC). While runtime systems like StarPU dynamically schedule tasks using directed acyclic graphs (DAGs) to optimize execution, their complex queue management heuristics make performance analysis increasingly difficult. Traditional profiling tools, heavily reliant on timeline-based Gantt charts, often become visually overwhelming at scale and struggle to provide an application-oriented perspective, hindering the diagnosis of performance bottlenecks. This dissertation proposes a two-stage visual analytics methodology to address this diagnostic gap. We first establish an iteration-centric paradigm that reorients performance profiles by mapping task execution metrics to algorithmic iterations, validating its diagnostic expressiveness under diverse scheduling and hardware constraints. Building upon these foundations, we introduce the k-hills visualization, a condensed 2D visualization technique that shifts the analytical focus from individual task lifespans to the aggregated computational volume applied across algorithmic iterations. By mapping the fraction of hardware dedicated to each task type over time, the visualization establishes a geometric column sum rule that instantly exposes hardware utilization and dependency-induced stalls. To validate the proposed methodology, we conduct a case study evaluating a dense tiled QR factorization on a 32-thread architecture. The study contrasts two dynamic scheduling policies: Work Stealing (WS) and Locality Work Stealing (LWS). While macroscopic metrics indicate a significant performance degradation for LWS, they cannot explain its root cause. The k-hills visualization successfully decodes this anomaly, visually proving that LWS falls into a locality trap—anchoring workers to heavy, older matrix updates and stalling the critical path. Conversely, the visualization demonstrates how WS actively unrolls the DAG, maintaining parallel throughput and preventing mid-execution starvation. Additionally, we compare StarPU and OpenMP signatures to show how the methodology exposes distinct scheduling behaviors and localized implementation anomalies. Ultimately, this work provides HPC developers with an intuitive, algorithm-aware diagnostic tool capable of translating abstract scheduler mechanics into actionable visual signatures.

Keywords: Task-Based Parallelism. Performance Visualization. Visual Analytics. Dynamic Scheduling. High-Performance Computing. Tiled QR Factorization. Dense Linear Algebra. StarPU. StarVZ. Execution Traces.

CONTENTS

LIST OF FIGURES **7**
LIST OF TABLES **10**
LIST OF ABBREVIATIONS AND ACRONYMS **11**
1 INTRODUCTION **12**
2 THEORETICAL BACKGROUND **14**
2.1 Parallel Computing and Hardware Architectures **14**
2.2 The Task-Based Parallelism Paradigm **17**
2.3 Runtime Systems and Task Scheduling **20**
2.4 Target Workload: Tile QR Factorization **22**
2.5 Tracing Task-Aware Application Behavior **24**
3 RELATED WORK **25**
3.1 Timeline and Structural Representations **25**
3.2 Statistical and Aggregated Profiling **27**
3.3 Programmatic Analysis Frameworks **27**
3.4 Algorithm-Aware Visualization **28**
4 FOUNDATIONS OF ITERATION-CENTRIC VISUALIZATION **32**
4.1 Foundational Iteration-Centric Perspectives **32**
4.1.1 Gantt Chart 32
4.1.2 Gantt Variant 33
4.1.3 Subdivided K-Iteration Views 34
4.2 Experimental Setup and Validation Scenarios **36**
4.2.1 Hardware Infrastructure 37
4.2.2 Software Environment 37
4.2.3 Experimental Design 38
4.2.4 Statistical Methodology 38
4.2.5 Establishing the Performance Baseline 39
4.2.6 Insight 1: Task Prioritization 41
4.2.7 Insight 2: Diagnosing Hardware Bottlenecks via RAM Restrictions 43
5 THE K-HILLS VISUALIZATION: FORMULATION AND CASE STUDY VALIDATION **47**
5.1 The k-hills Visualization **48**
5.2 Case Study Setup **49**
5.2.1 Hardware and Software Environment 50
5.2.2 Workload and Parameter Tuning 50
5.2.3 Target of Analysis: Scheduling Policies 51
5.3 Analysis of Performance and the Starvation Symptom **51**
5.4 Visual Diagnosis: Unveiling the Bottleneck **53**
5.5 Diagnostic Proof of Concept: Exposing Implementation Anomalies **55**
6 CONCLUSION AND FUTURE WORK **59**
6.1 Synthesis of Results and Achievement of Objectives **59**
6.2 Future Work **60**
REFERENCES **62**
APPENDIX A — RESUMO ESTENDIDO **65**

LIST OF FIGURES

Figure 2.1 Execution timelines illustrating the fundamental difference between concurrent (interleaved execution to hide latency) and parallel (simultaneous physical execution) models. The colors represent tasks to be executed. 16

Figure 2.2 Execution timelines illustrating the idle time induced by global barriers in traditional parallelism compared to the dense, overlapping execution achieved in task-based models. 17

Figure 2.3 A Directed Acyclic Graph (DAG) representing the execution flow and data dependencies of a tile-based factorization, categorized by iteration step (k). 19

Figure 2.4 Step-by-step update of the matrix tiles during a single iteration k of the tile QR factorization. 23

Figure 3.1 Standard taxonomy of a Gantt chart representation, mapping discrete events and data dependencies onto hardware resource tracks over continuous time (SAKIN; ISAACS, 2024) 25

Figure 3.2 A topological DAG representation (adapted from DAGViz (HUYNH et al., 2015)), emphasizing hierarchical task dependencies and execution flow while abstracting away the physical timeline. 26

Figure 3.3 A composite visualization generated via StarVZ's programmatic assembly (NESI et al., 2020). The figure demonstrates vertical panel composition (synchronizing Iteration, Gantt, Submitted, and Ready task panels on a shared time axis) and horizontal trace composition (comparing LWS and DMDAS scheduling policies side-by-side). 28

Figure 3.4 Execution comparison of an Elimination Tree traversal between LWS (left) and PRIO (right) schedulers (NESI et al., 2020). The plot highlights how scheduling decisions alter communication patterns and affect available tree-level parallelism. 29

Figure 3.5 The Cholesky Iteration plot mapping execution time (x-axis) against loop numbers (y-axis). The representation illustrates the task presence across iterations but highlights the severe visual occlusion caused by overlapping simultaneous tasks. 30

Figure 4.1 Comparison of Gantt charts for the Draco machine with hyperthreading disabled, illustrating the impact of tile size on resource utilization. On the left, a tile size of 1000 leads to suboptimal resource utilization, where the system fails to reach its full computational regime. On the right, a tile size of 100 achieves optimal capacity utilization, where the isolated empty spaces in thread 0 correspond to the creation and submission of tasks following the Sequential Task Flow (STF). 33

Figure 4.2 The first ten iterations for the CEI machine with hyperthreading disabled and a block size of 100. The visualization highlights how `DLARFB` tasks progressively consume a larger portion of the iteration's computational power as the iteration index increases. 34

Figure 4.3 The first four iterations for the CEI machine with hyperthreading disabled and a tile size of 100, demonstrating the subdivision of each y-axis iteration row into distinct sub-rows for individual task types. 35
Figure 4.4 An abstracted view of the first four iterations for the CEI machine with hyperthreading disabled and a tile size of 100. This variant renders a single bounding bar per task type, bounded by the start time of the first task and the end time of the last task of that type within the given iteration. 36
Figure 4.5 Mean makespan across all evaluated machines and tile sizes. Results with hyperthreading enabled are depicted in blue, while results with hyperthreading disabled are shown in red. Error bars represent a 99% confidence interval. 40
Figure 4.6 Execution traces from the Cei machine (tile size of 200, iterations 9 to 20). The top charts display bounding bars for `DLARFB` tasks, annotated with the total temporal span from the first task's start to the last task's completion, while other tasks are faded. The bottom charts reveal the individual `DLARFB` task executions, demonstrating the dispersion and overlap under different threading configurations. 42
Figure 4.7 Comparative makespan results across architectures. The chart contrasts the optimized `normal` baseline against variants applying `DLARFB` priority (`prio`) and strict synchronization (`taskwait`), evaluated at both optimal and fine-grained (100) tile sizes. 43
Figure 4.8 Iteration traces contrasting the Beagle and Draco machines utilizing a tile size of 400 with *hyperthreading* disabled. The x-axis represents execution time. 44
Figure 4.9 Comparative makespan analysis of the Beagle machine against the Draco machine under standard and memory-limited configurations. The optimal mean makespan (in seconds) for each variant is annotated above the respective blue markers. 45
Figure 4.10 Execution traces from the Draco machine (200 tile size, *hyperthreading* disabled, iterations 9 to 20) under constrained RAM. The top charts display bounding bars for `DLARFB` tasks annotated with their total temporal span, with other tasks faded. The bottom charts reveal the individual `DLARFB` task executions, demonstrating the dispersion and overlap induced by memory limitations. 46

Figure 5.1 Conceptual schematic of the k-hills visualization using generic kernels. The vertical distribution across iterations and the column sum demonstrate how system resources are partitioned over time. 48
Figure 5.2 Scheduling efficiency of WS vs. LWS. After iteration 10, LWS prioritizes cache locality, forming vertically dense execution blocks that delay the critical path and cause starvation (red). Conversely, WS aggressively steals critical panel tasks, forming a faster-advancing diagonal that unrolls the DAG, preventing worker idleness and significantly reducing makespan. 54

Figure 5.3 Algorithmic progression signatures showing the first three iterations under different runtime configurations: (a) StarPU implementation from 5 featuring the sequential `dlarfb` dependency, (b) the corrected parallel `dlarfb` StarPU configuration, and (c) OpenMP baseline execution..... 56

LIST OF TABLES

Table 4.1 Hardware specifications of the nodes utilized in the experimental evaluation. 37
Table 4.2 Experimental configurations that exhibit non-normal distributions according to the Shapiro-Wilk test ($p < 0.01$). 39

LIST OF ABBREVIATIONS AND ACRONYMS

BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDR4	Double Data Rate 4
FLOPs	Floating-Point Operations Per Second
GPU	Graphics Processing Unit
HPC	High-Performance Computing
INF	Instituto de Informática
LAPACK	Linear Algebra PACKage
LPPD	Parallel and Distributed Processing Laboratory
LWS	Locality Work Stealing
NUMA	Non-Uniform Memory Access
PCAD	High-Performance Computing Park
RAM	Random Access Memory
SMT	Simultaneous Multithreading
UFRGS	Federal University of Rio Grande do Sul
WS	Work Stealing

1 INTRODUCTION

The cessation of Dennard scaling and the physical limitations imposed by the power wall have fundamentally altered microprocessor design, driving the industry toward multi-core and many-core architectures (ESMAEILZADEH et al., 2011). To achieve continuous performance improvements, modern software must explicitly utilize parallel execution paradigms. However, as hardware complexity increases with the proliferation of non-uniform memory access (NUMA) topologies and chiplet-based designs, traditional loop-level parallelism suffers from severe load imbalance and interconnect bottlenecks.

The task-based parallelism paradigm, relying on directed acyclic graphs (DAGs) and asynchronous execution, has emerged as a robust solution to these challenges. By decoupling work definition from execution, runtime systems can dynamically schedule tasks to optimize hardware utilization. Despite these advancements, achieving optimal performance remains a highly complex mapping problem. Schedulers must constantly balance the competing demands of data locality—minimizing costly data transfers across memory nodes—and active load balancing across available processing cores, making performance analysis at the hardware level highly relevant and increasingly difficult.

While parallel programming models/runtimes like OpenMP (BOARD, 2008) and StarPU (AUGONNET et al., 2009) provide sophisticated abstractions for task-based execution, their underlying scheduling heuristics react differently to diverse hardware topologies, making comprehensive performance analysis an open research problem. Traditional profiling tools focus on identifying individual task bottlenecks, heavily relying on timeline-based trace views that obscure macroscopic scheduling patterns. In highly asynchronous, out-of-order environments, isolating these individual anomalies fails to capture the systemic behavior of the runtime. Addressing this requires an unconventional analytical approach. Building upon the pioneering work of the StarVZ (GARCIA PINTO et al., 2018) performance analysis framework, this dissertation argues for a shift in focus: rather than isolating micro-level bottlenecks, analysis must emphasize macro-level execution patterns and the inherent mechanics of the runtime system.

The primary objective of this dissertation is to assess the feasibility of generating robust performance hypotheses by examining the interaction between task submission, execution times, and the iteration that submitted it. Rather than focusing on individual bottleneck identification, this work prioritizes the comparative analysis of performance categorized by task type. In order to accomplish this, this work introduces a new, improved

version of the k-iteration chart that is already present in the StarVZ framework. This chart is specifically designed to illustrate the dynamics of task submission and execution.

To systematically evaluate and demonstrate the efficacy of this new tool, this study utilizes a tile QR factorization of a dense linear matrix (KURZAK et al., 2010) as the foundational workload. Because the computational weight and data dependencies of these tiles are highly uniform, this algorithm provides a controlled, predictable execution environment. Consequently, any observed execution anomalies can be directly attributed to the scheduler's mapping decisions or the underlying hardware topology, rather than algorithmic irregularities. This isolation allows for a clear, methodological validation of the k-iteration chart before applying it to more complex workflows.

The specific contributions of this work are threefold. First, it maps task submission iteration versus execution time by conceptualizing and implementing specific visualization enhancements within the StarVZ framework. Second, it provides a methodological validation of this unconventional analytical approach using a highly regular mathematical workload. Finally, it provides a comprehensive, hardware-level analysis of the efficacy of OpenMP and StarPU schedulers in managing dense QR factorization in the various topologies of the High-Performance Computing Park (PCAD) at the UFRGS Parallel and Distributed Processing Laboratory (LPPD).

The remainder of this dissertation is organized as follows. Chapter 2 establishes the theoretical background, detailing parallel computing paradigms and the microarchitectural characteristics that dictate execution efficiency. Chapter 3 reviews the related work in performance visualization, highlighting the limitations of traditional timeline tools and framing the analytical gap this research addresses. Chapter 4 formally introduces the iteration-centric analysis paradigm, establishing the use of algorithmic iterations as the foundational baseline for performance tracking. Chapter 5 details the formulation and visual construction of the k-hills visualization, presenting the empirical case study and validating the methodology under different runtime environments and scheduling policies within a tiled QR factorization. Finally, Chapter 6 synthesizes the research findings, confirms the achievement of the proposed objectives, and outlines promising directions for future work.

2 THEORETICAL BACKGROUND

This chapter provides the necessary theoretical foundations to understand the challenges and solutions explored in this dissertation. We establish the fundamental concepts of parallel computing, followed by an in-depth exploration of the task-based parallelism paradigm, programming models, and the microarchitectural characteristics that dictate execution efficiency in modern hardware.

2.1 Parallel Computing and Hardware Architectures

Historically, the continuous performance scaling of software applications relied almost exclusively on exponential increases in the clock frequency of single-core central processing units (CPUs). This era of free performance scaling was sustained by Dennard scaling (ESMAEILZADEH et al., 2011), a physical scaling law positing that as transistors shrink, their power density remains constant, allowing for higher clock frequencies without a disproportionate increase in power consumption. However, as transistor dimensions approached atomic limits, leakage currents and extreme heat dissipation caused Dennard scaling to break down. This thermal threshold—widely recognized in computer science as the *power wall*—forced a definitive paradigm shift in microprocessor design. Instead of pursuing unsustainable frequency increases, manufacturers pivoted to integrating multiple processing cores onto a single silicon die, establishing the multi-core era. Consequently, to sustain performance gains, mainstream software engineering was forced to move beyond purely sequential execution. Although parallel programming paradigms long predated the multi-core transition in high-performance computing, this architectural shift rendered concurrent and parallel execution essential for general-purpose applications seeking to exploit modern multi-core and distributed hardware resources.

To formally understand these execution models, one must establish the operating system abstractions that manage them. As detailed by Tanenbaum and Bos in *Modern Operating Systems* (TANENBAUM; BOS, 2015), the operating system manages execution primarily through the concepts of processes and threads. A process constitutes an independently schedulable execution container, encapsulating a program's compiled code, its runtime state, and a strictly isolated virtual memory address space. Because of this rigid isolation, data exchange between distinct processes requires inter-process communication

(IPC) mechanisms—such as pipes, sockets, or message passing—which incur substantial computational overhead and latency due to kernel-space context switching.

To mitigate this overhead, the thread abstraction was introduced. A thread is the smallest sequence of programmed instructions that can be managed independently by the operating system scheduler. Multiple threads residing within a single process share the exact same memory address space and system resources. This shared-memory architecture enables extremely high-bandwidth, low-latency communication between execution streams, forming the foundational software architecture for modern shared-memory parallel programming. However, concurrent access to shared data necessitates explicit, often complex synchronization mechanisms—such as mutexes, semaphores, or atomic operations—to prevent race conditions and maintain memory consistency.

Within the domain of high-performance computing (HPC), it is imperative to rigorously distinguish between concurrency and parallelism. While often used interchangeably in general software development, they describe related but fundamentally distinct execution semantics. Concurrency is a software design property referring to the logical structuring of a program into independent tasks that can be interleaved or executed out of order without altering the final outcome. It involves managing multiple streams of execution to hide latency; for instance, while one thread is blocked waiting for a high-latency disk input/output (I/O) operation, the scheduler context-switches to another thread to keep the CPU utilized. Concurrency can be fully realized on a single-core processor.

Parallelism, conversely, is a hardware utilization property referring to the strictly simultaneous execution of multiple computations across distinct physical processing units to achieve absolute reductions in execution time (makespan). While concurrency is about dealing with multiple things at once, parallelism is about doing multiple things at the exact same physical instant.

The efficacy with which multithreaded software can exploit physical parallelism is profoundly dictated by the underlying memory architecture. In early symmetric multiprocessor (SMP) systems, Uniform Memory Access (UMA) topologies were prevalent. In a UMA system, all processing cores share physical memory through a centralized interconnect or memory bus. The defining characteristic of this topology is that memory access latency and bandwidth remain constant regardless of which core initiates the request or which memory bank is targeted. While conceptually straightforward for parallel software development, UMA introduces severe contention and scalability bottlenecks. As core counts increase, the centralized memory bus quickly saturates, creating a “memory wall”

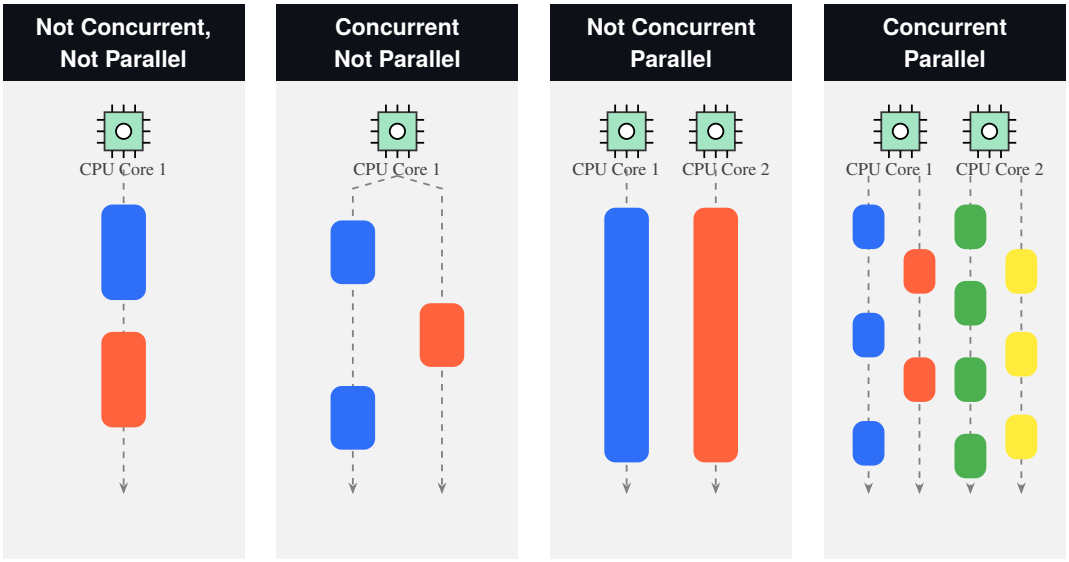


Figure 2.1 – Execution timelines illustrating the fundamental difference between concurrent (interleaved execution to hide latency) and parallel (simultaneous physical execution) models. The colors represent tasks to be executed.

that stifles further performance gains.

To transcend the scalability limitations of the centralized memory bus, modern HPC architectures—such as the Intel Xeon and AMD EPYC/Threadripper processors utilized in contemporary computing clusters—universally adopt Non-Uniform Memory Access (NUMA) topologies, as extensively described by Hennessy and Patterson in *Computer Architecture: A Quantitative Approach* (HENNESSY; PATTERSON, 2011). In a NUMA architecture, physical memory is logically partitioned and distributed across the system, with specific memory banks tightly coupled to specific physical processor sockets or internal chiplets (nodes).

When a processor accesses memory located within its local NUMA node, it achieves maximal bandwidth and minimal latency. Conversely, when a processor must fetch data from memory attached to a remote node, the request must traverse inter-processor interconnects (such as Intel's Ultra Path Interconnect or AMD's Infinity Fabric). This remote access significantly increases latency, diminishes available bandwidth, and introduces cache-coherency overhead across the system. Consequently, maximizing parallel efficiency on NUMA hardware is no longer just a matter of dividing computation evenly; it demands explicit data locality management. Parallel runtimes and schedulers must possess the architectural awareness to actively map computational workloads to the specific physical cores residing in the same NUMA node as the required data. This critical requirement exposes the limitations of traditional loop-based parallelism and directly motivates the adoption of advanced, data-driven task scheduling paradigms.

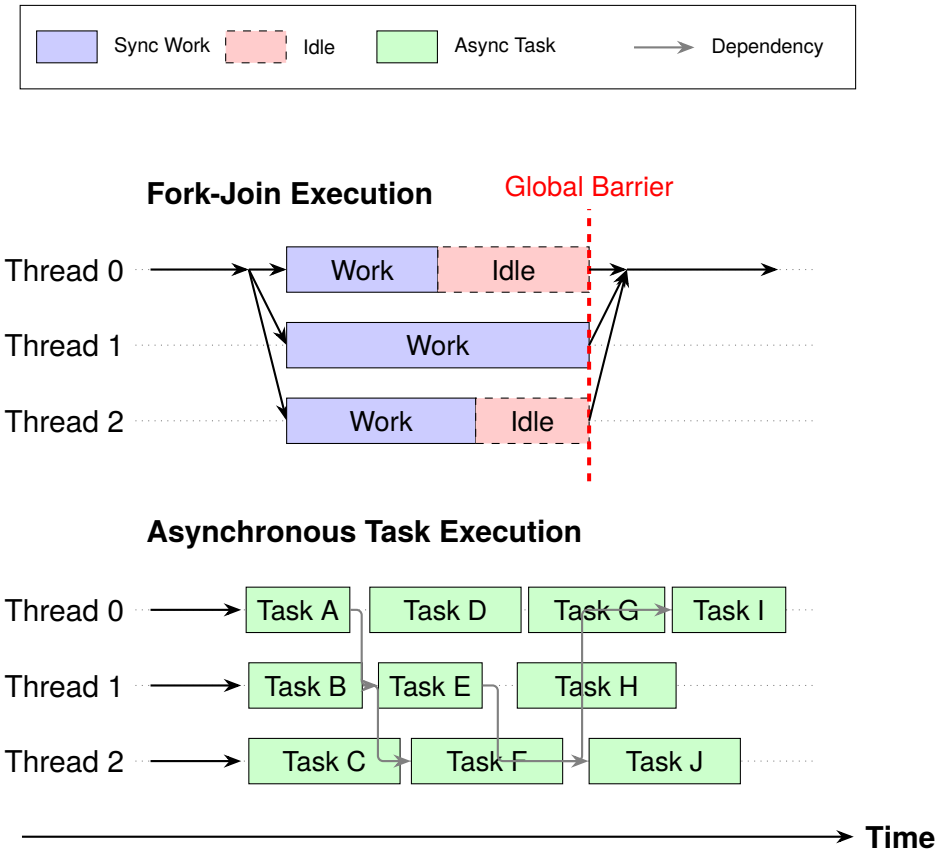


Figure 2.2 – Execution timelines illustrating the idle time induced by global barriers in traditional parallelism compared to the dense, overlapping execution achieved in task-based models.

2.2 The Task-Based Parallelism Paradigm

Historically, shared-memory parallel programming relied heavily on loop-level parallelism, predominantly implemented via the highly structured fork-join execution model. In this paradigm, a master thread spawns a team of worker threads to process independent loop iterations in parallel. While highly effective for strictly regular workloads with uniform computational costs, this approach introduces severe performance bottlenecks in modern, complex environments due to its reliance on rigid global synchronization barriers.

As illustrated in the top timeline of Figure 2.2, if loop iterations entail varying computational costs—or if threads experience disparate memory access latencies inherent to NUMA topologies—faster threads complete their assigned work prematurely. Consequently, these threads are forced into an idle state at the global synchronization barrier, squandering valuable processor cycles while awaiting the slowest thread. This structural load imbalance directly limits the theoretical maximum speedup of the application. The conceptual constraints of the fork-join model are clearly reflected in traditional OpenMP directives, as contrasted with modern approaches in Listing 2.1.

```
// --- Variant 1: Fork-Join Execution ---
// Threads split iterations; implicit global barrier at end.
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++) {
    process_work(i);
}

// --- Variant 2: Asynchronous Task Execution ---
// Master thread generates tasks; workers execute
   concurrently without explicit dependencies.
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < N; i++) {
        #pragma omp task
        process_task(i);
    }
}
```

Listing 2.1 – Pseudo-code: Fork-Join vs. Task-based OpenMP execution models.

To resolve these synchronization bottlenecks and accommodate increasingly irregular algorithms, the high-performance computing community has increasingly explored alternative execution models, driving significant interest toward the task-based parallelism paradigm. This model fundamentally decouples the definition of computational work from its mapping to physical execution resources. A *task* is defined as a cohesive, self-contained unit of computational work explicitly coupled with the specific data environment it requires to operate. Rather than manually mapping instructions to specific threads, developers decompose the application into logical tasks and submit them to a sophisticated underlying *runtime* system. The *runtime's scheduler* assumes the responsibility of dynamically distributing these tasks across available hardware resources, allowing the application to inherently adapt to the underlying hardware topology and mitigate load imbalance without manual intervention. A concrete example of this paradigm is illustrated in Figure 2.3, which delineates the task-based decomposition of a tile QR factorization algorithm across consecutive iterations (k). The representation highlights how the global workload is par-

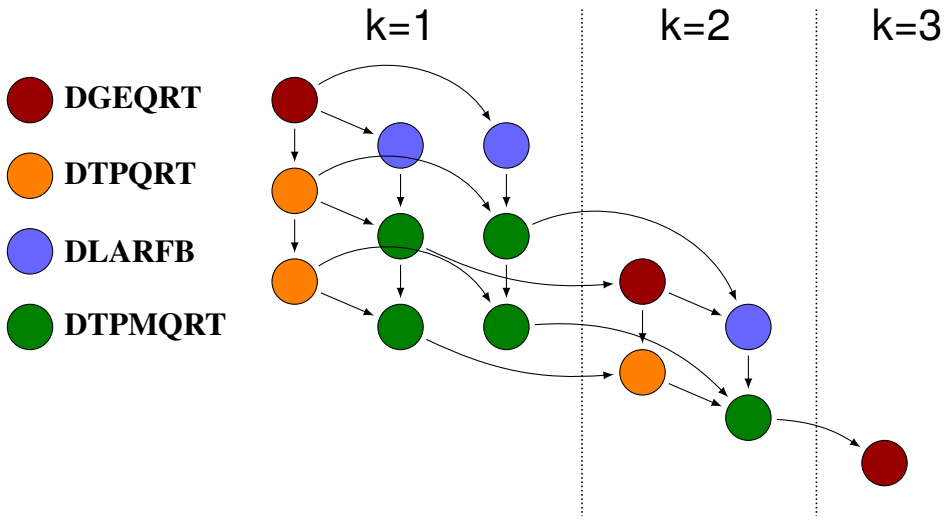


Figure 2.3 – A Directed Acyclic Graph (DAG) representing the execution flow and data dependencies of a tile-based factorization, categorized by iteration step (k).

titioned into distinct, specialized task types that must be dynamically orchestrated by the scheduler.

To enforce algorithmic correctness without resorting to costly global barriers, the task-based model governs execution order through fine-grained data dependencies. Upon task creation, the programmer explicitly specifies the target memory regions and their required access modes. Read-only access allows multiple tasks to consume the same data concurrently, whereas write or read-write access demands strictly exclusive control over the memory region to prevent race conditions. By continuously evaluating these access patterns, the *runtime* system dynamically constructs a Directed Acyclic Graph (DAG), as depicted in Figure 2.3. In practice, this graph functions as a clear execution roadmap where individual tasks are represented as nodes (the colored circles) and data dependencies as directed edges (the arrows). The term *directed* means that the execution flows strictly one way—an arrow from Task A to Task B dictates that Task B cannot begin until Task A completes. Concurrently, the term *acyclic* guarantees that the network contains no loops, ensuring that the workflow always moves forward toward completion and preventing the scheduler from getting trapped in infinite execution cycles.

Within this graph structure, vertices represent the individual computational tasks (e.g., the specific mathematical *kernels* in a matrix factorization), and directed edges denote the critical data dependencies across successive algorithm iterations. For instance, if one task generates a memory block required by a subsequent task, a Read-After-Write (RAW) dependency exists. The *runtime* establishes a directed edge between them, guaranteeing that the consumer task cannot be scheduled until the producer task has successfully

committed its results to memory.

The formulation of the execution flow as a DAG enables an entirely asynchronous, non-sequential execution. The *runtime* system continuously tracks task readiness through these dynamic data relationships, placing tasks into a ready queue the exact moment all their incoming dependencies are satisfied. This asynchronous flow inherently overlaps computation with data movement. While specific cores execute ready tasks, the *runtime* can prefetch necessary data across the NUMA interconnect for pending tasks. Devoid of rigid global barriers, the system processes tasks in a continuous, densely packed pipeline. As demonstrated in the bottom timeline of Figure 2.2, this data-driven execution model maximizes hardware utilization and exposes the theoretical maximum parallelism intrinsic to the algorithm's structure.

2.3 Runtime Systems and Task Scheduling

To translate theoretical task-based paradigms into executable software, developers rely on parallel programming models and their underlying *runtime* systems. These models provide the fundamental software abstractions necessary to define computational tasks, specify data dependencies, and manage memory. The *runtime* system then acts as an intermediary orchestration layer between the compiled application and the operating system, managing the Directed Acyclic Graph (DAG) and dynamically mapping tasks to available hardware resources. Within this ecosystem, OpenMP and StarPU represent two distinct and prominent paradigms for task management.

Open Multi-Processing (OpenMP) (BOARD, 2008) is the widely adopted, industry-standard application programming interface (API) for shared-memory parallelism. While initially conceived for traditional fork-join loop parallelism, OpenMP evolved significantly to provide robust support for task-based execution and fine-grained data dependencies. OpenMP expresses parallelism primarily through declarative compiler directives (*pragmas*). Developers designate a specific code block as a task utilizing the `#pragma omp task` directive. Upon encountering this directive, the executing *thread* packages the code block and its associated data environment into a task structure, deferring its execution to a *thread* pool managed by the OpenMP *runtime*. Dependencies are explicitly declared via the `depend` clause, which specifies the required access mode (`in`, `out`, or `inout`) and the target memory addresses. The *runtime* dynamically monitors these addresses to construct the DAG on the fly, dispatching tasks only when the requisite memory regions are safe

to access. Additionally, OpenMP provides mechanisms for explicit synchronization, such as the `#pragma omp taskwait` directive, which strictly blocks the encountering *thread* until all of its locally generated child tasks have completed.

In contrast to OpenMP's directive-based standard, StarPU (AUGONNET et al., 2009) is a highly specialized *runtime* system engineered via an explicit C library API. Originally developed to exploit heterogeneous architectures combining CPUs and hardware accelerators, StarPU provides a sophisticated execution engine that is equally powerful in homogeneous NUMA environments. The fundamental abstraction in StarPU is the `Codelet`, a data structure that defines the computational *kernel* and specifies its available hardware implementations. A defining characteristic of StarPU is its centralized, rigid data management system. Before memory can be accessed by a task, it must be explicitly registered with the *runtime* to generate a `Data Handle`. Because the *runtime* strictly controls these handles alongside access mode flags (`STARPU_R`, `STARPU_W`, or `STARPU_RW`), it maintains absolute global awareness of data placement. This enables StarPU to automatically enforce cache coherency across NUMA nodes, moving data asynchronously prior to execution and feeding ready tasks to its pluggable task schedulers.

While the programming model dictates how tasks and dependencies are defined, the *task scheduler* dictates exactly when and where these tasks execute. The *scheduler* functions as the central nervous system of the *runtime*, dynamically mapping the DAG onto the physical hardware to minimize the overall execution time (*makespan*). Schedulers resolve this mapping problem through specific policies, broadly categorized into static and dynamic approaches. Static scheduling assigns tasks to processing units strictly prior to execution. Although this minimizes mapping overhead and yields predictable execution traces, it is inherently fragile, frequently inducing severe load imbalance when processing irregular workloads or encountering unpredictable system noise.

Conversely, dynamic scheduling assigns tasks at *runtime* as processing resources become idle. A universally adopted dynamic policy is *work-stealing* (BLUMOFE; LEISERSON, 1999). In this decentralized model, each worker *thread* maintains a local queue of ready tasks. In StarPU, when a *thread* exhausts its own queue, it actively steals tasks from the queue of a busy peer. This strategy minimizes locking contention on centralized data structures and naturally balances the workload on the fly.

However, basic load distribution is insufficient for modern architectures. Advanced schedulers must also optimize for topological priority and data locality. Priority-based algorithms evaluate the DAG to assign execution weights, ensuring that tasks located on

the critical path are executed first, thereby preventing downstream execution starvation. Furthermore, in NUMA environments where data movement is often more computationally expensive than the mathematical operation itself, locality-aware schedulers become mandatory. These algorithms aggressively map tasks to the specific physical cores where the required data already resides in local memory banks. This strategy actively prioritizes data proximity over pure computational load balancing, reducing memory access latency and preventing interconnect saturation.

Despite their theoretical benefits, advanced mapping heuristics inherently introduce *scheduling overhead*—the processor cycles consumed by the *runtime* to compute mapping decisions rather than executing actual user code. If a *runtime* utilizes an overly complex algorithm to schedule extremely fine-grained tasks, the algorithmic decision overhead can easily eclipse the actual computation time. Consequently, achieving optimal parallel performance requires a delicate balance: the application must generate tasks with sufficient granularity to justify the computational expense of the *scheduler's* locality and priority algorithms, without creating chunks so large that they limit parallel concurrency.

2.4 Target Workload: Tile QR Factorization

To rigorously evaluate the efficacy of task scheduling policies, this study utilizes the tile-based QR factorization of a dense matrix as the foundational workload. Mathematically, the QR factorization decomposes a given matrix A into an orthogonal matrix Q and an upper triangular matrix R, such that $A = QR$. In the context of modern task-based parallelism, the global matrix is partitioned into localized square blocks, or tiles, effectively decomposing the monolithic operation into a set of fine-grained, independent tasks.

The execution flow of a single iteration k is driven by four primary computational kernels, as illustrated in Figure 2.4 (KURZAK et al., 2010). Each kernel targets specific regions of the matrix and establishes a strict hierarchy of data dependencies:

- **DGEQRT(A_{kk}):** This kernel computes the local QR factorization of the diagonal tile. It is the most critical task in each iteration, acting as the root of all subsequent dependencies. It generates the Householder reflectors required by all other kernels in the same iteration.
- **DLARFB(A_{kj}):** For each tile in the current row k where $j > k$, this kernel applies

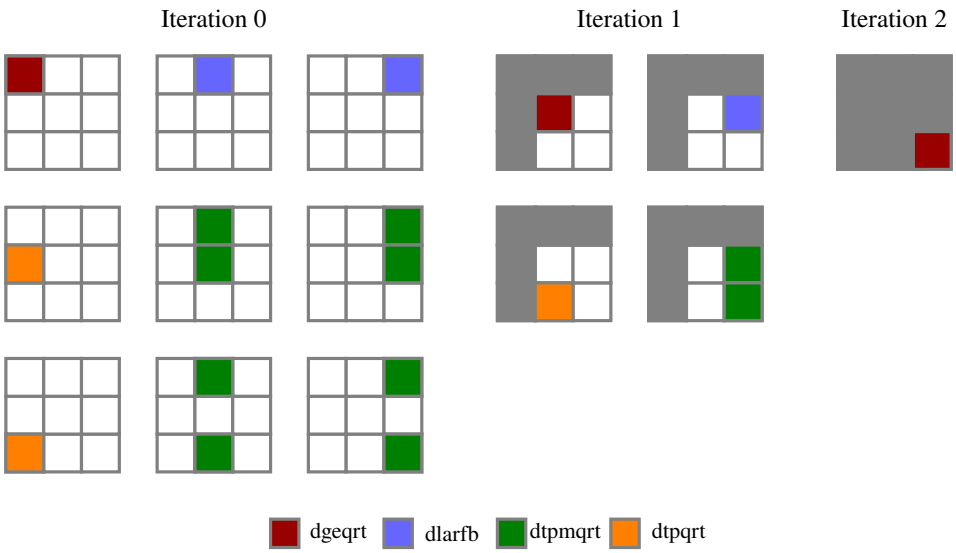


Figure 2.4 – Step-by-step update of the matrix tiles during a single iteration k of the tile QR factorization.

the transformations generated by `DGEQRT`. These tasks can execute in parallel with each other but are strictly blocked by the completion of the diagonal `DGEQRT`.

- **`DTPQRT`(A_{ik}):** For each tile in the current column k where $i > k$, this kernel updates the factorization using the triangular tile from the diagonal. Similar to `DLARFB`, these tasks are blocked by the diagonal `DGEQRT`.
- **`DTPMQRT`(A_{ij}):** These kernels update the remainder of the trailing matrix where $i, j > k$. A `DTPMQRT` task at position (i, j) has a dual dependency: it requires the updated reflectors from `DTPQRT` at (i, k) and the row data from `DLARFB` at (k, j).

The interaction between these kernels creates a complex Directed Acyclic Graph (DAG), as depicted in Figure 2.3. The bottleneck structure is clearly visible: the sparse, computationally light `DGEQRT` and `DTPQRT` tasks control the execution throughput of the numerous and computationally heavy `DTPMQRT` tasks.

This specific algorithmic structure is an ideal experimental vehicle. Because the mathematical workload of each kernel type is uniform, any observed execution anomalies—such as the gaps between `DLARFB` tasks or the stretching of the *makespan*—can be rigorously attributed to the runtime scheduler's inability to resolve these specific dependency blocks efficiently under hardware or memory constraints.

The implementation used in this work follows the algorithm from (KURZAK et al., 2010), which has been previously utilized in prior studies (MILETTO; SCHNORR, 2019). This code is applied across both Chapters 4 and Chapter 5.

2.5 Tracing Task-Aware Application Behavior

Tracing techniques record specific runtime events within a task-based application during execution. To accurately characterize application behavior, tracing must capture each task's start time, completion time, submitting iteration, and executing worker thread. The following describes how this data is extracted across different runtimes using tailored tracing strategies.

The OpenMP Tools Interface (OMPT) (EICHENBERGER et al., 2013) enables low-overhead runtime task tracking through user-defined callbacks. The runtime triggers notifications whenever specific state transitions occur, providing fine-grained insight into thread activities such as task execution, barrier entry, and barrier exit. At the time of writing, GCC lacks complete support for OMPT. Fortunately, LLVM/libomp provides all the functionality required for this dissertation, leading us to adopt the Clang compiler for OpenMP compilation. This work utilizes an in-house tracer developed at UFRGS. Because any tracing mechanism introduces performance overhead, keeping this impact minimal is crucial to isolate intrinsic application performance from tracer interference. A prior study evaluating the performance impact of this tracer was published in SSCAD 2025 (MATOS; SCHNORR, 2025). A custom script converts the resulting binary trace files into the format required by the StarVZ framework (GARCIA PINTO et al., 2018).

In contrast, StarPU generates execution traces directly via the FxT library—a binary format that logs all events occurring during program runtime. This binary trace is converted into the Pajé format using the starpu_fxt_tool utility and subsequently processed within the StarVZ framework for performance visualization and analysis.

3 RELATED WORK

This chapter reviews the state-of-the-art in performance visualization for parallel systems. We begin by examining traditional timeline and structural representations, highlighting their limitations regarding visual occlusion and algorithmic context. Next, we discuss statistical profiling tools and their inherent trade-offs concerning temporal data loss. Finally, we explore the recent shift toward programmatic analysis frameworks and algorithm-aware visualizations, identifying the specific analytical gap that this dissertation addresses.

3.1 Timeline and Structural Representations

Performance visualization of parallel systems traditionally relies on mapping discrete execution events across a horizontal temporal axis. Timeline-based plots, predominantly Gantt charts employed by profiling environments such as Paraver (COMPUTADORS et al., 1995) and the GPU-accelerated ViTE (COULOMB et al., 2011), effectively capture worker states, communication intervals, and temporal idleness. Figure 3.1 illustrates the standard taxonomy of these representations (SAKIN; ISAACS, 2024), where horizontal tracks represent hardware resources (e.g., CPU threads) and blocks denote computational tasks over time, often overlaid with dependency arrows.

While Gantt charts intuitively display temporal sequences, they suffer from catastrophic visual occlusion and high query latency when processing the massive event traces typical of high-performance computing. Millions of fine-grained tasks transform the visualization into an illegible expanse of microscopic blocks and overlapping dependency

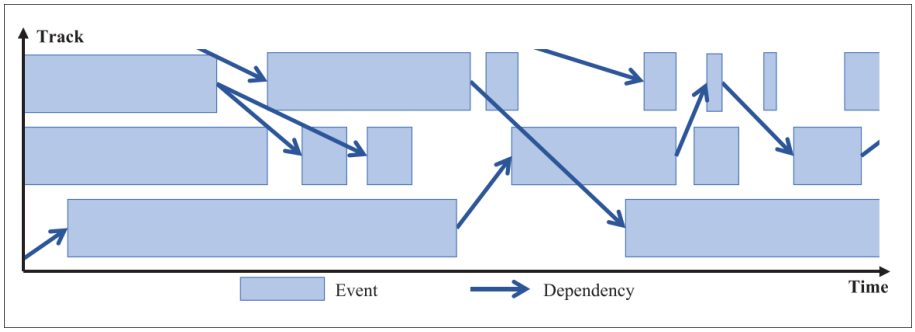


Figure 3.1 – Standard taxonomy of a Gantt chart representation, mapping discrete events and data dependencies onto hardware resource tracks over continuous time (SAKIN; ISAACS, 2024)

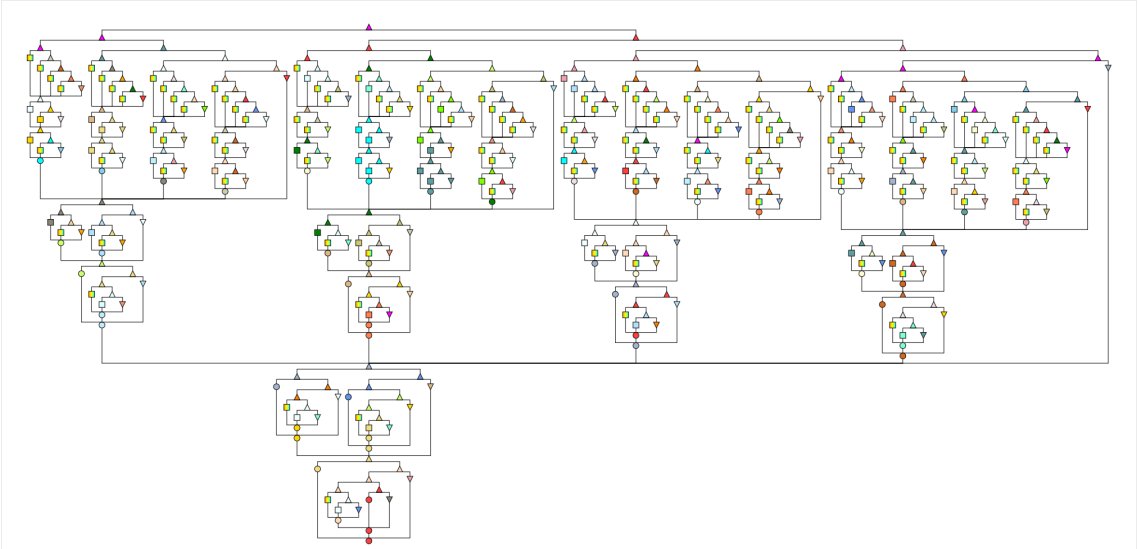


Figure 3.2 – A topological DAG representation (adapted from DAGViz (HUYNH et al., 2015)), emphasizing hierarchical task dependencies and execution flow while abstracting away the physical timeline.

lines. More critically, these representations remain semantically decoupled from the underlying mathematical application. A Gantt chart indicates *when* a physical resource is active, but it completely fails to convey *where* that execution is situated within the overall algorithmic progression, obfuscating the root cause of scheduling anomalies.

To expose this missing algorithmic structure, analysts traditionally pivot to Directed Acyclic Graphs (DAGs) to map task dependencies. Tools like DAGViz (HUYNH et al., 2015) enable the hierarchical exploration of these topological structures (Figure 3.2), whereas Grain Graphs manage visual complexity by aggregating recurring synchronization patterns, such as fork-join constructs (REISSMANN; MUDDUKRISHNA, 2018). Extending this structural perspective, frameworks such as Atria (WILLIAMS; BIGELOW; ISAACS, 2020) model execution as logical expression trees to highlight computational hotspots, and SkeVU (ERNSTSSON; FRANKELL; KESSLER, 2025) maps algorithmic skeletons directly to source code to guide optimization heuristics.

However, DAG-based plots inherently compromise the temporal dimension. By abstracting time to emphasize pure topology, they obscure physical hardware constraints. A DAG demonstrates what *can* theoretically run in parallel, but it cannot quantify *when* tasks actually executed, nor can it precisely measure proportional resource utilization or idleness during specific temporal windows. Consequently, analysts are trapped in a fundamental representational dichotomy: Gantt charts provide physical time but sacrifice algorithmic context, whereas DAGs provide algorithmic context but sacrifice physical time.

3.2 Statistical and Aggregated Profiling

To circumvent the visual clutter and scalability bottlenecks inherent to timeline representations, statistical profiling tools employ extensive data aggregation. Frameworks such as HPCToolkit (ADHIANTO et al., 2010) and TAU (SHENDE; MALONY, 2006), alongside visualizations like Flame Graphs (GREGG, 2016), summarize execution by accumulating the total time spent within specific call paths or functions. While this approach successfully resolves visual occlusion and highlights overarching computational hotspots, it completely eradicates both the temporal sequence and the topological context of the execution. By reducing dynamic performance data to static summaries, aggregated profilers obscure the intricate interactions between task scheduling, synchronization delays, and algorithmic progression. Consequently, they fail to capture the transient, systemic behaviors that dictate the efficiency of asynchronous *runtime* environments.

3.3 Programmatic Analysis Frameworks

To overcome the analytical rigidity of traditional monolithic tools, recent literature exhibits a transition toward programmatic analysis frameworks. Environments like Pipit (BHATELE et al., 2024), Hatchet (BHATELE; BRINK; GAMBLIN, 2019), and notably StarVZ (GARCIA PINTO et al., 2018; NESI et al., 2020), decouple trace processing from fixed graphical user interfaces, delegating visualization directly to the broader data science ecosystem. This paradigm shift transforms performance visualization from a static, pre-compiled application into a set of scriptable, highly customizable components.

StarVZ exemplifies this approach by modularizing performance visualization into independent visual units known as *panels*. Rather than forcing analysts to navigate a predefined, generic dashboard, StarVZ allows users to programmatically generate isolated visual panels tailored to specific metrics—such as standard Gantt charts, algorithmic iteration plots, or statistical time-series tracking task submissions and ready queues.

The primary analytical advantage of this framework lies in panel composition. Analysts can programmatically coordinate these independent views to construct customized, multi-dimensional perspectives of the execution. By vertically stacking diverse panels, analysts can synchronize completely different metrics along a shared temporal axis (e.g., aligning worker states with the volume of ready tasks). Furthermore, as demonstrated in Figure 3.3, StarVZ enables complex comparative analysis through horizontal compo-

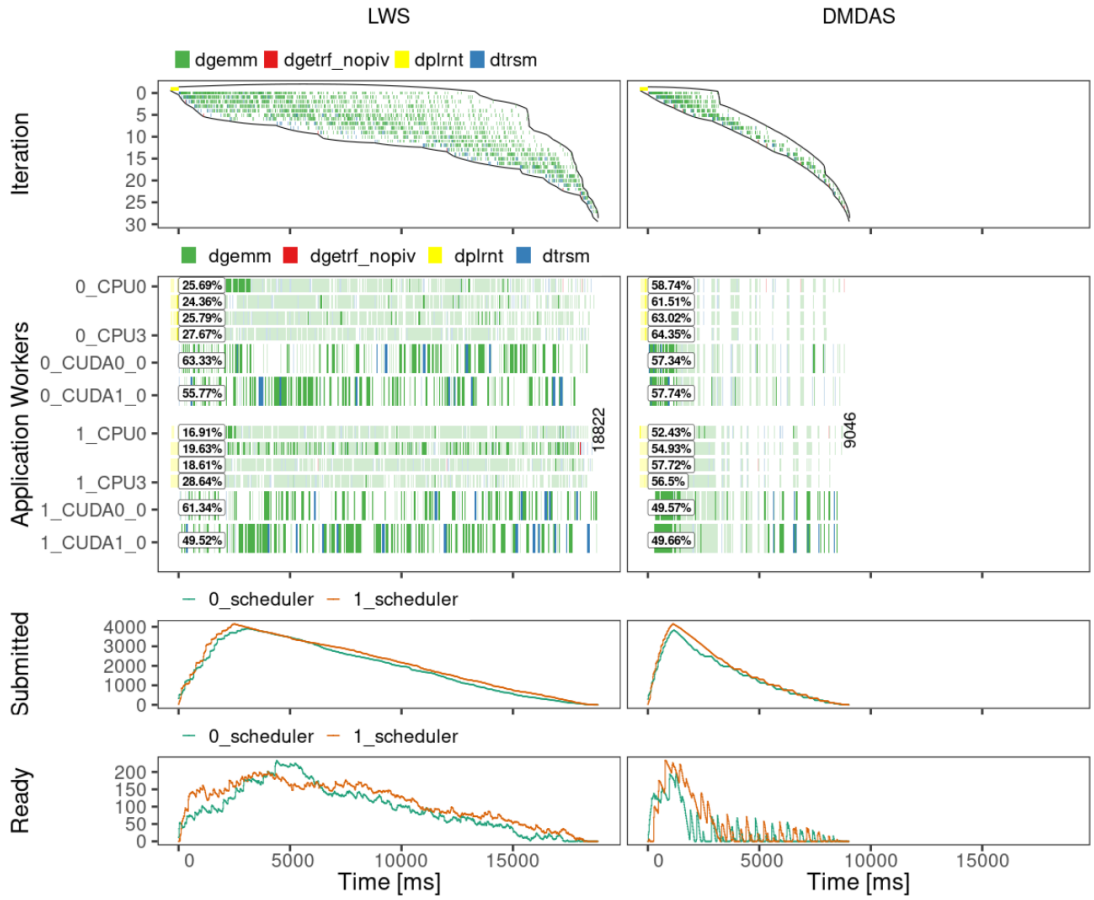


Figure 3.3 – A composite visualization generated via StarVZ's programmatic assembly (NESI et al., 2020). The figure demonstrates vertical panel composition (synchronizing Iteration, Gantt, Submitted, and Ready task panels on a shared time axis) and horizontal trace composition (comparing LWS and DMDAS scheduling policies side-by-side).

sition. Utilizing specific programmatic routines (e.g., `starvz_assemble`), analysts can project entirely distinct execution traces—such as those generated by different scheduling policies like LWS and DMDAS—side-by-side. The framework automatically standardizes the temporal axes, scales, and legends across the composition, eliminating visual redundancies and facilitating a rigorous, direct visual comparison of application behavior under varying configurations.

3.4 Algorithm-Aware Visualization

The modular flexibility of modern programmatic frameworks facilitates the creation of sophisticated, algorithm-aware panels that attempt to reconcile the temporal and structural perspectives of parallel execution. By mapping hardware events directly onto mathematical or topological application structures, these visualizations allow analysts to

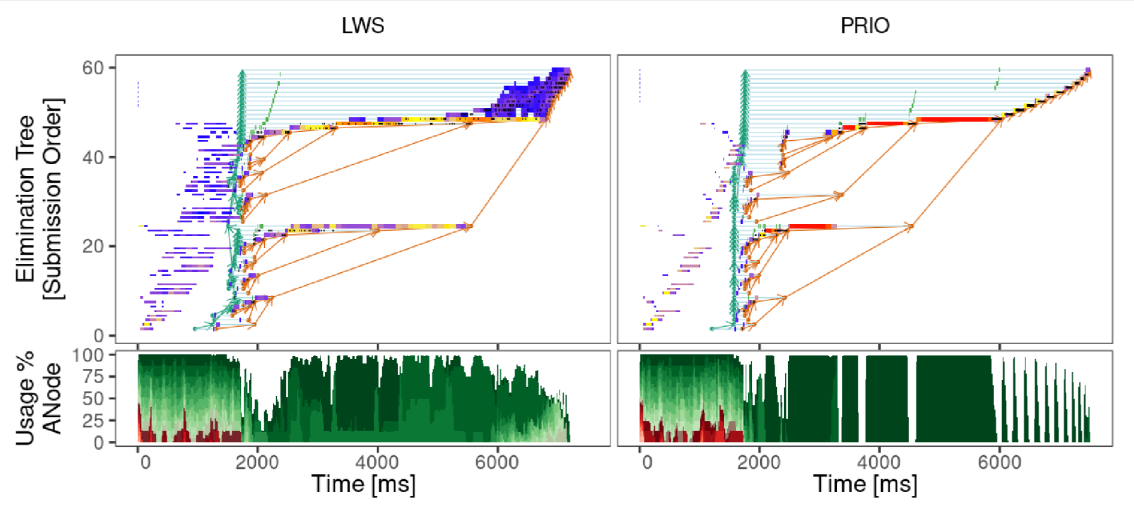


Figure 3.4 – Execution comparison of an Elimination Tree traversal between LWS (left) and PRIO (right) schedulers (NESI et al., 2020). The plot highlights how scheduling decisions alter communication patterns and affect available tree-level parallelism.

observe exactly how the *runtime* traverses the specific algorithm over time.

For instance, the Traveler framework (SAKIN et al., 2023) introduced the *Aggregated Gantt View* to elevate the level of abstraction beyond individual task intervals. Instead of rendering every microscopic task on an isolated track, this view aggregates operations into meaningful, logical blocks defined by the program's underlying execution tree. For example, if a specific primitive block of code resides within a loop running k times, the view aggregates the constituent fine-grained dependencies and generates k unified bars along the timeline. This approach provides a simplified temporal layout that correlates high-level algorithmic primitives over time, effectively reducing the visual clutter of a standard Gantt chart while maintaining chronological context.

Addressing highly irregular workloads, the *Elimination Tree* panel (MILETTO et al., 2022; NESI et al., 2020) was developed specifically to analyze task-based sparse matrix factorization solvers, such as QRMumps (AGULLO et al., 2013). These solvers utilize the Multifrontal method, decomposing the global sparse matrix into smaller, dense sub-problems known as frontal matrices. These matrices are topologically organized into an elimination tree structure. The visualization projects this algorithmic structure by mapping the tree nodes vertically (typically by submission order) against a horizontal, continuous time axis.

As demonstrated in Figure 3.4, this representation is highly effective for exposing how different *runtime* scheduling policies alter algorithmic traversal. When comparing executions side-by-side, an analyst can immediately observe that the priority-based (`PRIO`) scheduler delays communication tasks until the end of a node's execution block. This be-

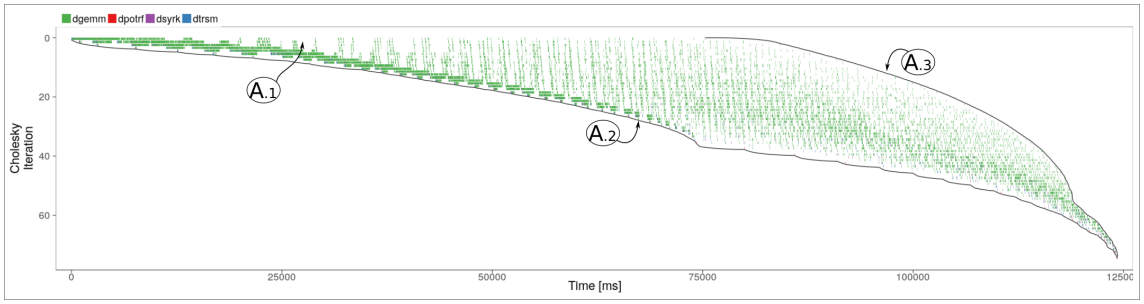


Figure 3.5 – The Cholesky Iteration plot mapping execution time (x-axis) against loop numbers (y-axis). The representation illustrates the task presence across iterations but highlights the severe visual occlusion caused by overlapping simultaneous tasks.

havior inadvertently bottlenecks parent nodes waiting for data, thereby limiting overall tree-level parallelism. In contrast, the locality-aware (`LWS`) scheduler interleaves communication with mathematical computation, unlocking broader parallel traversal of the tree.

Pushing this algorithmic alignment further for dense linear algebra, representations such as the *Cholesky Iteration* plot (GARCIA PINTO et al., 2018) directly map execution time (x-axis) against algorithmic loop iterations (y-axis), as seen in Figure 3.5. This creates a distinct application signature where tasks appear as color-coded glyphs bounded within their respective iterative steps.

While representations like the *Cholesky Iteration* plot successfully inject algorithmic context into the timeline, they inherit the fundamental flaw of continuous spatial mappings: severe visual occlusion. Because standard visualizations for task-based systems rely heavily on absolute temporal coordinates, analysts are forced into a rigid trade-off: accept overlapping tasks or abandon the structural context entirely. If one attempts to resolve this spatial overlap using standard statistical techniques—such as two-dimensional histograms or continuous area charts—the structural dimension inevitably collapses. Although these aggregations accurately quantify macroscopic resource utilization, they obliterate the granular algorithmic context, making it impossible to determine which specific mathematical iterations are occupying the processing units at any given moment.

Despite the robust extensibility of programmatic environments and recent advancements in hybrid panels, a critical analytical gap remains unresolved. Existing plots inherently struggle to simultaneously depict continuous execution time, algorithmic depth, and a strict, uniform quantification of proportional resource allocation without introducing visual clutter. Consequently, there is an explicit need for a novel visual methodology. To effectively audit modern scheduling heuristics, analysts require a representation that discretizes temporal execution against algorithmic progression using stacked, proportional

metrics. By completely eliminating spatial overlap while preserving both time and iteration depth, such an approach would empower analysts to instantly correlate critical-path delays with precise worker idleness across the entire lifecycle of the application.

4 FOUNDATIONS OF ITERATION-CENTRIC VISUALIZATION

Understanding *scheduler* behavior and execution decisions is a significant challenge in task-based parallel applications. This chapter introduces a methodological framework designed to analyze these interactions by shifting the analytical focus from physical resources to algorithmic progress. To establish a structured validation of this research, our contribution is organized into two sequential stages across Chapter 4 and Chapter 5. This chapter focuses on validating this foundational paradigm under the OpenMP runtime, utilizing the task submission iteration variable on the vertical axis as a complementary perspective to traditional profiling tools. Through this approach, we evaluate how alternative visual abstractions can expose scheduling bottlenecks and hardware constraints in operations such as tile QR factorization. Once this iteration-centric paradigm is validated under diverse experimental scenarios, Chapter 5 will introduce k-hills—our final condensed geometric visualization technique—alongside an advanced case study evaluating dynamic scheduler mechanics.

4.1 Foundational Iteration-Centric Perspectives

This section introduces a novel perspective for analyzing scheduler and application behavior, shifting the focus from hardware utilization toward an iteration-centric paradigm. We begin by reviewing the standard Gantt chart in subsection 4.1.1. Subsequently, subsection 4.1.2 presents our proposed iteration-based variant, and subsection 4.1.3 details the specific k-iteration visualizations developed for this methodological framework.

4.1.1 Gantt Chart

The traditional, resource-oriented Gantt chart serves as a fundamental instrument in performance analysis. By plotting tasks against computational resources on the y-axis over time on the x-axis, it facilitates the assessment of thread utilization and scheduling efficiency. Specifically, it reveals whether the application generates a sufficient volume of tasks to maintain the *OpenMP* runtime fully occupied. Suboptimal tile size configurations, for instance, manifest as visible idle gaps between executed tasks, as demonstrated on the

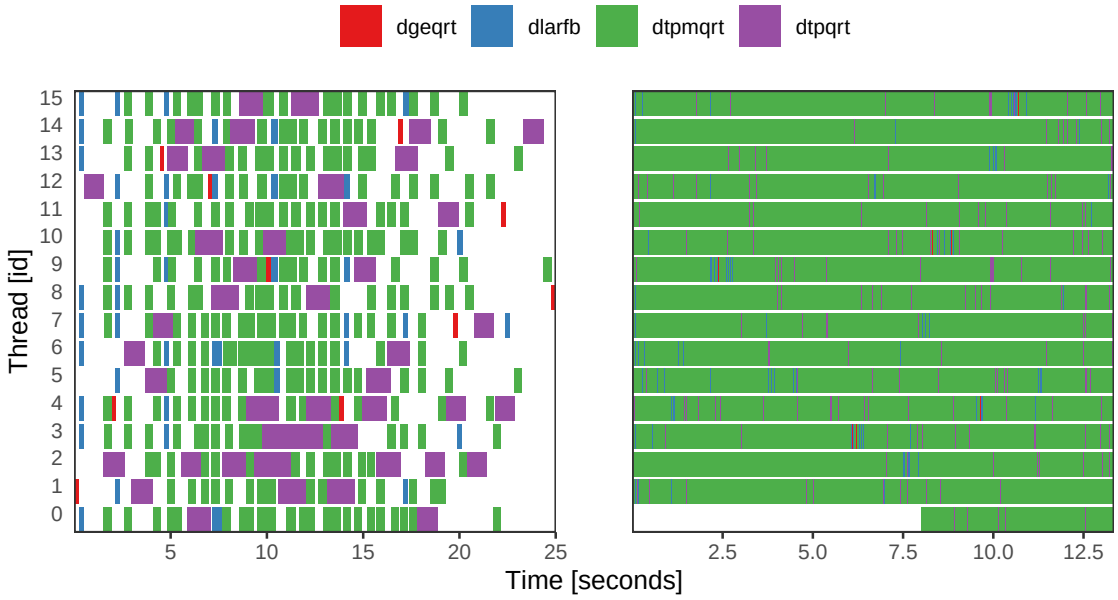


Figure 4.1 – Comparison of Gantt charts for the Draco machine with hyperthreading disabled, illustrating the impact of tile size on resource utilization. On the left, a tile size of 1000 leads to suboptimal resource utilization, where the system fails to reach its full computational regime. On the right, a tile size of 100 achieves optimal capacity utilization, where the isolated empty spaces in thread 0 correspond to the creation and submission of tasks following the Sequential Task Flow (STF).

left side of Figure 4.1. Conversely, an optimal tile size ensures dense resource utilization, characterized by a continuous stream of tasks with negligible idle time, as depicted on the right of Figure 4.1 (where idle periods are isolated exclusively to thread 0 during the Sequential Task Flow creation phase).

4.1.2 Gantt Variant

While traditional Gantt charts are highly effective, they are not perfect, and critical runtime dynamics can occasionally be overlooked. A prime example of this is the behavior of `DLARFB` tasks, whose progression patterns are easily masked in standard views. To address this, the proposed variant reorients the analytical axis by mapping the iteration of task creation to the y-axis rather than the physical computational resource (see Figure 4.2). Both the traditional Gantt chart and this iteration-based variant clearly demonstrate that `DTPMQRT` tasks dominate the execution time of each iteration. Furthermore, the proposed variant distinctly highlights dynamic shifts in proportional cost, revealing that `DLARFB` tasks gradually consume a larger fraction of the computational effort as the algorithm progresses into deeper iterations.

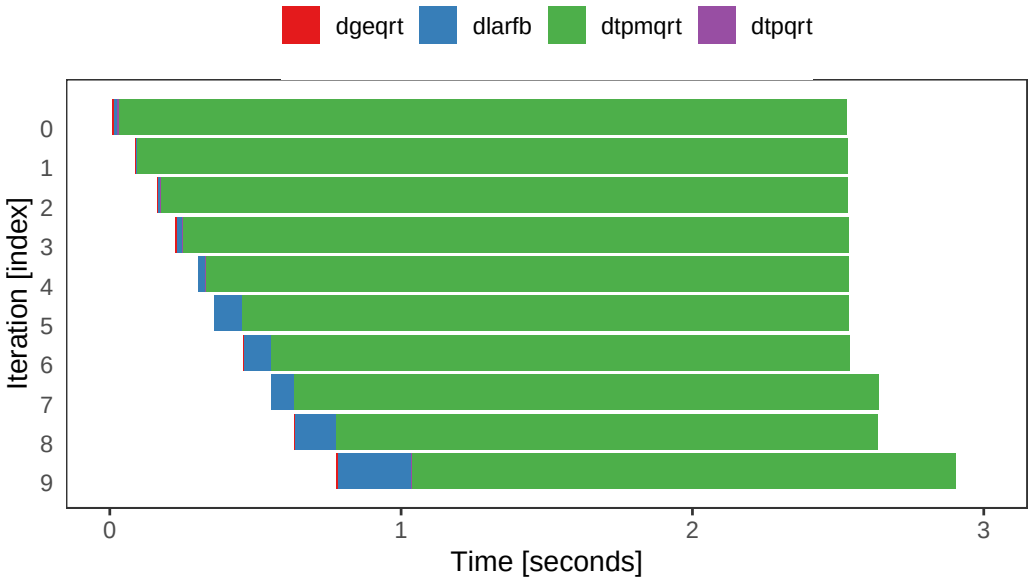


Figure 4.2 – The first ten iterations for the CEI machine with hyperthreading disabled and a block size of 100. The visualization highlights how `DLARFB` tasks progressively consume a larger portion of the iteration's computational power as the iteration index increases.

However, a significant limitation of this variant is its inability to effectively display tasks with extremely short execution times. Consequently, the `DGEQRT` and `DTPQRT` kernels are virtually imperceptible, as their corresponding bars are too slim to be visually resolved on the temporal x-axis. While this accurately reflects their negligible computational footprint, it inadvertently obscures their critical structural role as dependencies that block numerous subsequent tasks. This visual shortcoming—alongside the inherent occlusion caused by parallel task execution, which is only partially mitigated by applying transparency (opacity) to the task representations—necessitates a more granular analytical perspective, directly motivating the development of the k-iteration variants.

4.1.3 Subdivided K-Iteration Views

Building upon the k-iteration panel of StarVZ, this approach maps tasks on the y-axis corresponding strictly to their source iteration. Each iteration row is further subdivided into distinct sub-rows representing specific task types. For the tile QR factorization analyzed in this study, tasks are sequentially ordered top-to-bottom as `DGEQRT`, `DLARFB`, `DTPQRT`, and `DTPMQRT`, reflecting their logical progression within the algorithm's codebase. The x-axis continuously represents time.

Figure 4.3 illustrates the initial four iterations of this approach. This visualization effectively exposes structural dependencies and generates actionable hypotheses regarding

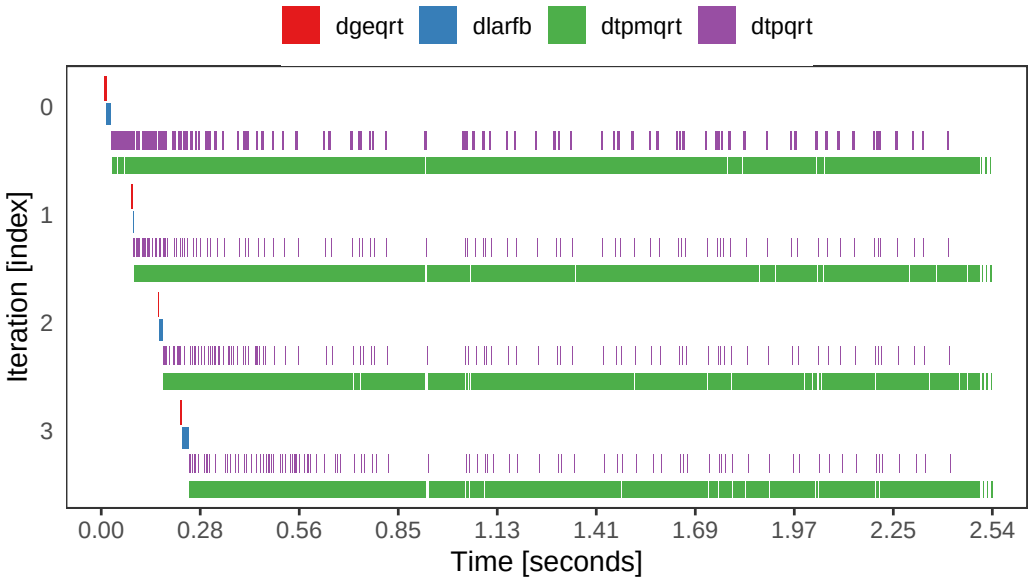


Figure 4.3 – The first four iterations for the CEI machine with hyperthreading disabled and a tile size of 100, demonstrating the subdivision of each y-axis iteration row into distinct sub-rows for individual task types.

runtime behavior. For example, observing that all `DLARFB` tasks strictly succeed `DGEQRT`, and `DTPQRT` tasks strictly succeed `DLARFB`, visually confirms the rigid dependency chain dictated by the algorithm. As with the Gantt variant, opacity is employed to mitigate visual overlap, where regions of higher color saturation intuitively indicate high degrees of concurrent execution.

An alternative abstraction of this chart collapses the individual task executions into a single, continuous bounding bar per task type per iteration. In this summarized view, the bar spans from the absolute start time of the first task to the absolute end time of the final task of that specific type within the iteration. As illustrated in Figure 4.4, this macroscopic perspective facilitates the identification of macro-correlations. For instance, cross-referencing Figure 4.2 and Figure 4.3 confirms that `DTPMQRT` consumes the bulk of computational resources.

Incorporating the abstracted view (Figure 4.4) further exposes the intricate dependency dynamics governing the `DTPQRT` and `DTPMQRT` phases. As observed in Figure 4.3, `DTPQRT` tasks appear sparse because they algorithmically block themselves, forming a strict sequential dependency chain. Each individual `DTPQRT` task then acts as a gatekeeper, blocking multiple `DTPMQRT` tasks. While `DLARFB` tasks also block `DTPMQRT`, their highly parallel execution minimizes their impact as a structural bottleneck. Consequently, despite the algorithm spawning a significantly larger volume of `DTPMQRT` tasks, their execution throughput is fundamentally dictated by the sequential resolution of the sparse

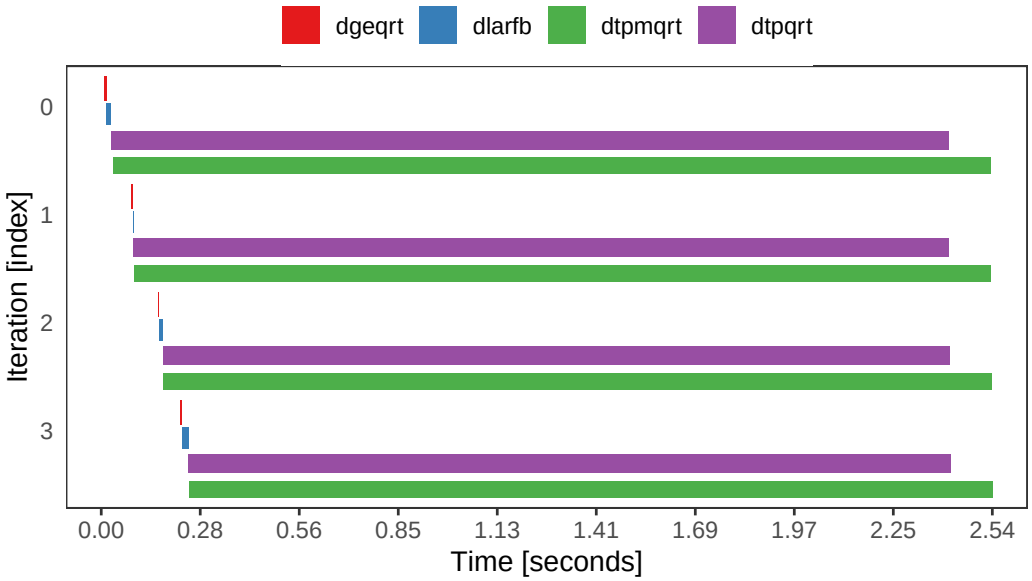


Figure 4.4 – An abstracted view of the first four iterations for the CEI machine with hyperthreading disabled and a tile size of 100. This variant renders a single bounding bar per task type, bounded by the start time of the first task and the end time of the last task of that type within the given iteration.

`DTPQRT` kernels. This firmly establishes `DTPQRT` as the primary bottleneck in this execution phase. Finally, this abstracted view clearly depicts that subsequent iterations merely undergo a temporal shift, suggesting that inter-iteration dependencies exert a relatively low penalty on the overall *makespan*.

4.2 Experimental Setup and Validation Scenarios

To rigorously evaluate the diagnostic expressiveness and practical utility of the proposed iteration-centric paradigm, a comprehensive set of empirical experiments was designed. Rather than serving as a conventional performance validation, these scenarios are specifically engineered as analytical benchmarks to assess how effectively the iteration-on-y axis perspective can expose diverse behavioral anomalies that traditional resource-centric views struggle to capture. This section details the computational environment, software stack, and the structured experimental phases executed to generate the distinct execution traces analyzed throughout our diagnostic insights.

4.2.1 Hardware Infrastructure

The empirical evaluation was conducted utilizing the High-Performance Computing (HPC) facility provided by the Informatics Institute at the Federal University of Rio Grande do Sul (INF/UFRGS). The testbed comprises four heterogeneous nodes, each equipped with different generations of Intel Xeon processors. This diversity in core counts, memory capacity, and underlying motherboard architectures ensures that the visualizations are tested across varied architectural behaviors. Table 4.1 details the specific hardware configurations of each machine.

Table 4.1 – Hardware specifications of the nodes utilized in the experimental evaluation.

Machine	**CPU Configuration**
cei	2 x Intel(R) Xeon(R) Silver 4116, 2.10 GHz, 48 threads, 24 cores
draco	2 x Intel(R) Xeon(R) E5-2640 v2, 2.00 GHz, 32 threads, 16 cores
beagle	2 x Intel(R) Xeon(R) E5-2650, 2.00 GHz, 32 threads, 16 cores
hype	2 x Intel(R) Xeon(R) E5-2650 v3, 2.30 GHz, 40 threads, 20 cores

4.2.2 Software Environment

To guarantee a controlled and reproducible execution environment, all experiments were hosted on Debian GNU/Linux 12, utilizing libc version 2.36. The compilation and task orchestration relied on the Clang compiler (version 18.1.8) acting as the OpenMP runtime. The underlying linear algebra operations were supported by OpenBLAS version 0.3.27, provisioned via the Spack package manager. To ensure mathematical correctness and address specific library defects, LAPACK was manually built from source using the commit hash `6ec7f2bc4ecf4c4a93496aa2fa519575bc0e39ca`, which incorporates custom bug fixes. Furthermore, the OpenMP library was explicitly installed via the Spack package `llvm-openmp` version 18.1.0.

4.2.3 Experimental Design

The evaluation methodology was structured into three distinct experimental phases, each targeting a specific analytical scenario using a dense matrix of size 10000×10000 elements of double-precision floating-point numbers.

Phase 1: Baseline and Tile Size Optimization. The initial phase aimed to establish a baseline and determine the optimal block size for each architecture. We evaluated an array of tile sizes: 100, 125, 200, 250, 400, 500, 625, 1000, and 1250. Every configuration was executed across all four machines under two threading paradigms (hyperthreading enabled and disabled). Each unique variant underwent 20 independent replications to ensure statistical stability. The overarching *makespan* results for these runs are summarized in Figure 4.5 and thoroughly discussed in subsection 4.2.5.

Phase 2: Dependency Prioritization and Synchronization. The second phase focused on observing how the scheduler reacts to manual code interventions. Confined to the Beagle and Draco machines, this phase introduced two distinct scheduling manipulations: (1) assigning elevated execution priority explicitly to `DLARFB` tasks, and (2) enforcing a strict synchronization barrier using the `#pragma omp taskwait` directive immediately following `DLARFB` task submissions. These interventions were tested against two tile sizes: the empirically determined optimal size for each machine (400 for Draco and 125 for Beagle) and a uniform baseline size of 100. Hyperthreading was disabled, and 20 replications were performed per variant.

Phase 3: Memory Constraint Emulation. The final phase was engineered to isolate and investigate hardware discrepancies. We constrained the Draco machine to a maximum of 30 GB of RAM via the Slurm directive `#SBATCH -mem=30G`, artificially matching the physical memory limits of the Beagle machine. We subsequently re-executed the comprehensive tile size sweep from Phase 1 on both Beagle and Draco (with hyperthreading toggled on and off, executing 20 replications per setup) to observe memory-bound performance degradation, later analyzed in subsection 4.2.7.

4.2.4 Statistical Methodology

To ensure the statistical rigor of our comparative analyses, the normality of the execution time distributions was initially evaluated using the Shapiro-Wilk test. For the vast majority of the experimental configurations—categorized by machine, hyperthreading

Table 4.2 – Experimental configurations that exhibit non-normal distributions according to the Shapiro-Wilk test ($p < 0.01$).

Machine	Tile Size	Hyperthreading	Priority	Taskwait	Memory Limited	p-value
cei	125	Disabled	No	No	No	0.008
cei	125	Enabled	No	No	No	0.004
cei	250	Enabled	No	No	No	< 0.001
draco	200	Disabled	No	No	No	< 0.001
draco	400	Disabled	No	No	No	< 0.001
draco	400	Disabled	No	Yes	No	0.002
draco	500	Disabled	No	No	No	0.002
hype	100	Disabled	No	No	No	0.002

state, and tile size—the null hypothesis could not be rejected, indicating normal distributions. Consequently, the independent two-sample t-test is employed by default to compare mean *makespans* throughout this study.

However, seven specific variants exhibited statistically significant deviations from normality, leading to the rejection of the null hypothesis (as cataloged in Table 4.2). In these exceptional cases, the non-parametric Mann-Whitney U test is utilized for comparative analysis, which will be explicitly noted in the text when applicable. All confidence intervals rendered in the visualizations are calculated using a strict 99% confidence level.

4.2.5 Establishing the Performance Baseline

Before applying the proposed iteration-centric visualizations, an initial experimental phase was conducted to establish a performance baseline and determine the optimal tile size for each architecture. As depicted in Figure 4.5, the mean *makespan* for each machine is evaluated across varying tile sizes, contrasting configurations with *hyperthreading* enabled (blue) and disabled (red). The empirical data reveals distinct architectural responses to logical threading.

For the Beagle machine, the results clearly demonstrate that disabling *hyperthreading* consistently yields superior performance across all evaluated tile sizes. The optimal *makespan* was achieved utilizing a tile size of 125, recording 14.07 ± 0.40 seconds. This represents a statistically significant improvement over the baseline tile size of 100 (14.43 ± 0.22 seconds), corroborated by an independent t-test yielding a p-value of 0.001252, strictly satisfying our 99% confidence threshold.

A different behavioral pattern emerges when analyzing the Cei and Hype machines.

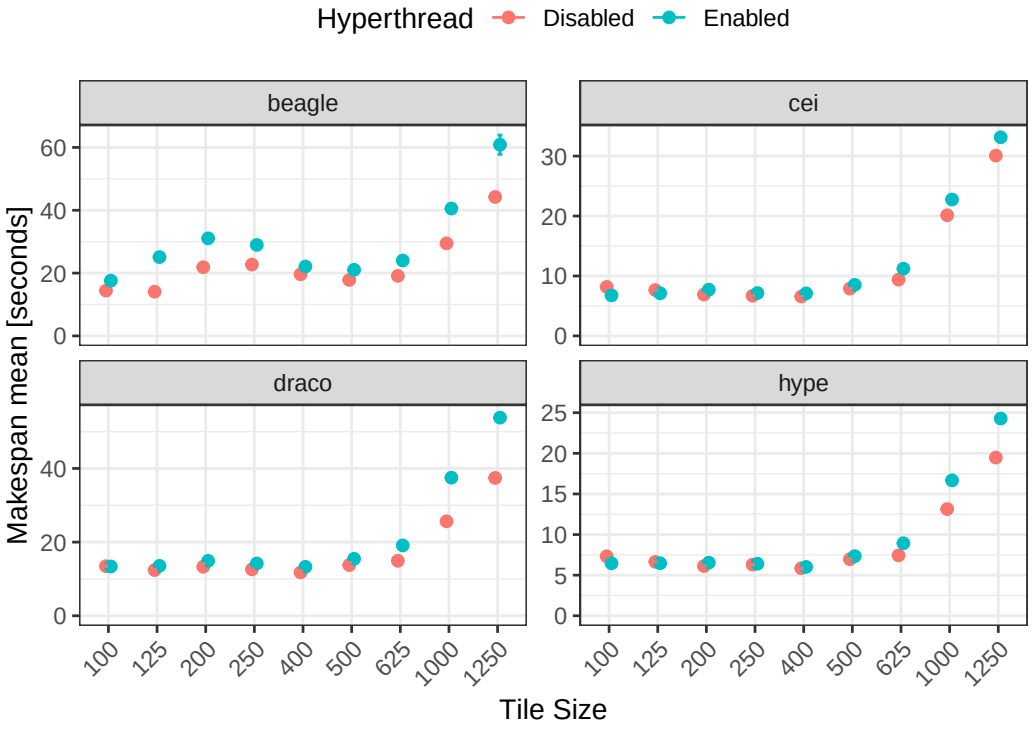


Figure 4.5 – Mean makespan across all evaluated machines and tile sizes. Results with hyperthreading enabled are depicted in blue, while results with hyperthreading disabled are shown in red. Error bars represent a 99% confidence interval.

In these architectures, enabling *hyperthreading* proved advantageous exclusively at the smallest tile size (100). For intermediate configurations, performance between the two threading modes remained statistically analogous until reaching a tile size of 625, beyond which disabling *hyperthreading* established a clear performance advantage. Specifically, the Cei machine reached its absolute optimal performance using a tile size of 400 with *hyperthreading* disabled, achieving a mean *makespan* of 6.57 ± 0.03 seconds. This configuration statistically outperforms the best *hyperthreading*-enabled scenario (tile size of 100, yielding 6.76 ± 0.02 seconds), as confirmed by a t-test ($p < 0.001$).

The Draco node exhibited a preliminary trend similar to Cei and Hype at a tile size of 100; however, the Mann-Whitney U test produced a p-value of 0.60. Consequently, we cannot reject the null hypothesis, indicating no statistically significant benefit to enabling *hyperthreading* at this granularity. For all subsequent, larger tile sizes, disabling *hyperthreading* definitively accelerated execution. Peak performance on the Draco architecture was recorded at a tile size of 400, achieving a mean *makespan* of 11.77 ± 0.66 seconds.

In summary, the empirical analysis identifies a unified optimal configuration for the Cei, Hype, and Draco nodes: a tile size of 400 coupled with disabled *hyperthreading*. Conversely, the Beagle machine requires a distinct parameterization, reaching its absolute

minimum *makespan* with a tile size of 125 and *hyperthreading* disabled. With these optimal baseline metrics established, the subsequent case studies leverage the proposed visual methodology to investigate the underlying *scheduler* behaviors and dependency dynamics that drive these performance outcomes.

4.2.6 Insight 1: Task Prioritization

A comparative analysis of the iteration charts exposes a critical scheduling variance induced by logical threading. As illustrated in Figure 4.6 for the Cei machine, isolating the `DLARFB` executions reveals that enabling *hyperthreading* significantly elongates the temporal span required to complete these tasks within a given iteration. By examining the detailed rendering of individual `DLARFB` tasks, it becomes visually evident that *hyperthreading* causes these tasks to scatter over the timeline, introducing substantial idle gaps. Conversely, disabling *hyperthreading* results in a denser, more cohesive execution block, suggesting the OpenMP *scheduler* inherently processes the `DLARFB` phase more efficiently under physical-core isolation.

To empirically validate whether this delayed resolution constitutes a structural bottleneck, two targeted *scheduler* interventions were tested: explicitly assigning high priority to `DLARFB` tasks (`prio`), and enforcing a strict synchronization barrier via `#pragma omp taskwait` immediately following their submission (`taskwait`). These interventions were evaluated using the empirically optimal tile sizes (400 for Draco and 125 for Beagle) and a smaller baseline size (100). The smaller tile size was specifically included to artificially inflate the iteration count, thereby amplifying the potential penalty of cumulative scheduling delays.

Figure 4.7 summarizes the performance impacts of these interventions. Explicitly prioritizing `DLARFB` tasks successfully mitigated the observed scheduling dispersion, yielding statistically significant *makespan* reductions on the Draco and Hype architectures. In contrast, while the baseline (`normal`) variant nominally outperformed the prioritized variant on the Cei and Beagle machines, an independent t-test yielded p-values of 0.263 and 0.5938, respectively, indicating no statistically significant degradation.

Ultimately, the proposed visual methodology successfully diagnosed a subtle scheduling inefficiency. Translating this visual insight into a direct code optimization—task prioritization—definitively enhanced performance on half of the evaluated architectures without incurring a statistically significant penalty on the remainder.

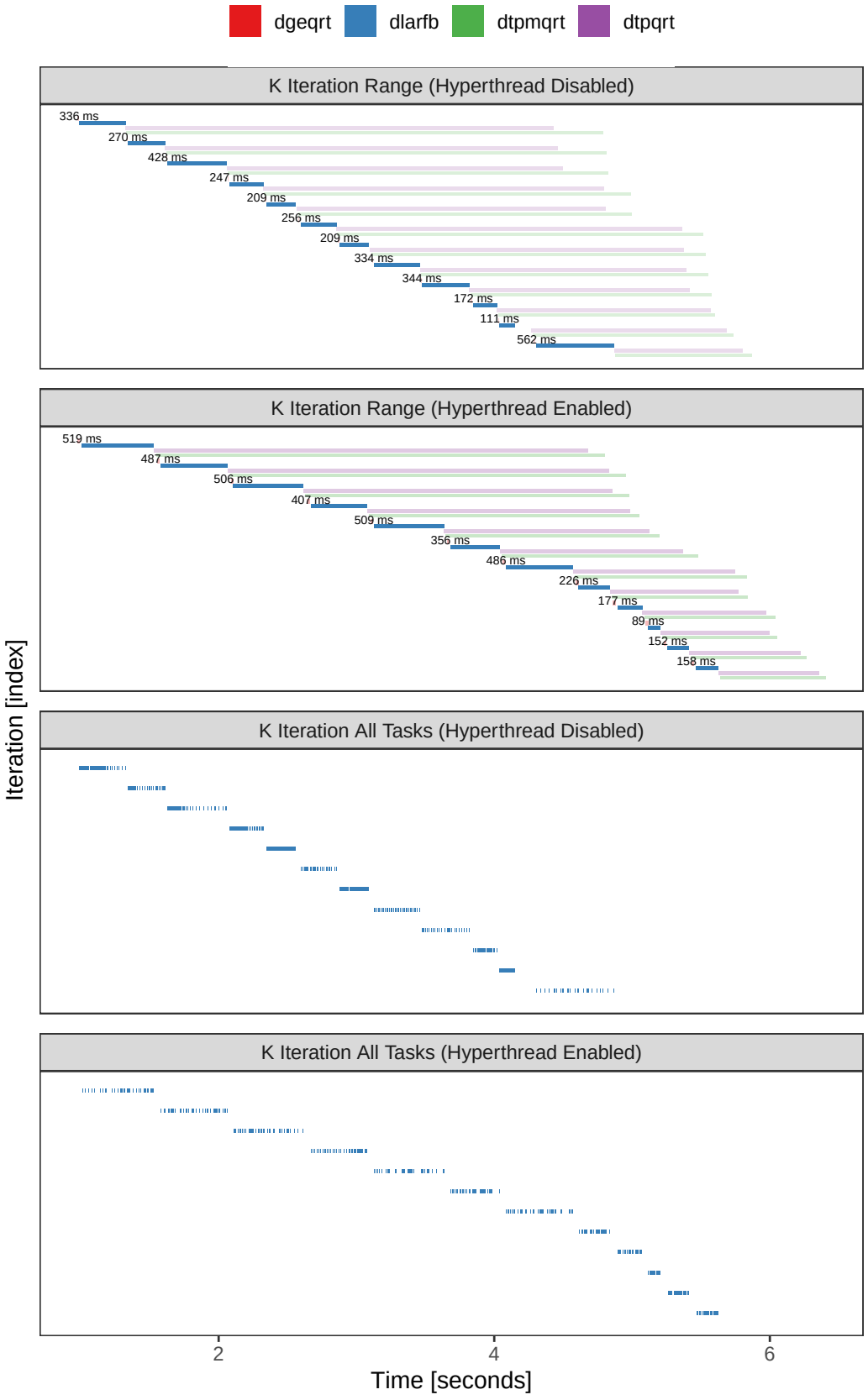


Figure 4.6 – Execution traces from the Cei machine (tile size of 200, iterations 9 to 20). The top charts display bounding bars for `DLARFB` tasks, annotated with the total temporal span from the first task's start to the last task's completion, while other tasks are faded. The bottom charts reveal the individual `DLARFB` task executions, demonstrating the dispersion and overlap under different threading configurations.

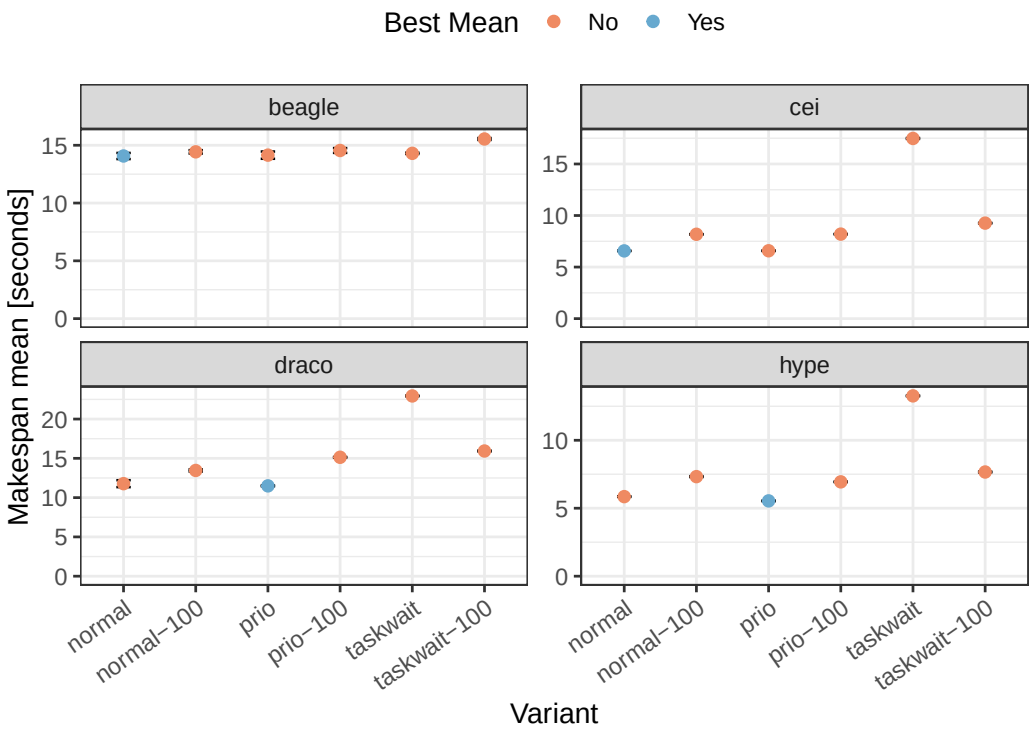


Figure 4.7 – Comparative makespan results across architectures. The chart contrasts the optimized `normal` baseline against variants applying `DLARFB` priority (`prio`) and strict synchronization (`taskwait`), evaluated at both optimal and fine-grained (100) tile sizes.

4.2.7 Insight 2: Diagnosing Hardware Bottlenecks via RAM Restrictions

The Beagle architecture consistently plateaued at approximately 15 seconds across all configurations, underperforming relative to the Draco node despite their superficial hardware similarities. Beagle is equipped with dual Intel Xeon E5-2650 processors (8 cores, 2.00 GHz base, 2.80 GHz turbo), while Draco utilizes dual Intel Xeon E5-2640 v2 processors (8 cores, 2.00 GHz base, 2.50 GHz turbo). Although Beagle possesses a higher turbo frequency, Draco benefits from a newer microarchitecture (v2) that provides superior Instructions Per Clock (IPC) and memory subsystem efficiency. As illustrated in the iteration charts (Figure 4.8), Draco systematically outpaces Beagle at every individual iteration. This granular visual evidence eliminates gross *runtime* scheduling errors and isolates the degradation to a per-iteration hardware processing deficit, directly motivating the subsequent investigation into their respective memory constraints.

To identify the root cause, we hypothesized that the bottleneck stemmed from memory constraints rather than computational capacity. While Draco possesses 64 GB of RAM, Beagle is limited to 32 GB. To empirically test this, we artificially restricted Draco's available memory to 30 GB using the Slurm workload manager directive (`#SBATCH`

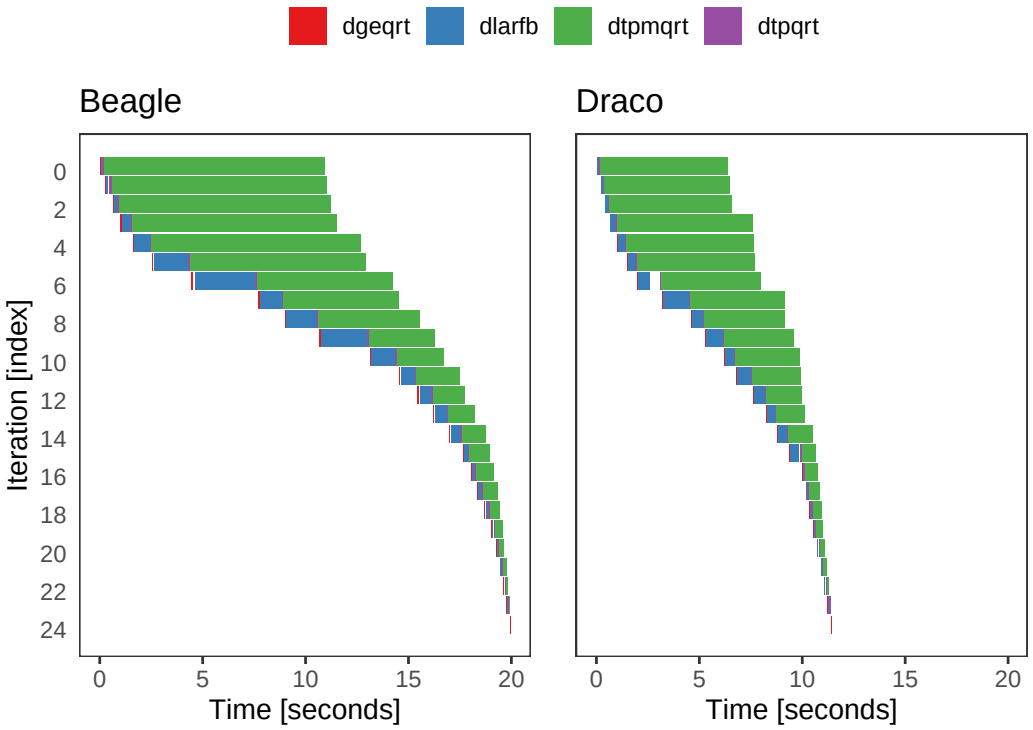


Figure 4.8 – Iteration traces contrasting the Beagle and Draco machines utilizing a tile size of 400 with *hyperthreading* disabled. The x-axis represents execution time.

`-mem=30G`), forcing a constrained operating environment comparable to Beagle.

Figure 4.9 contrasts the baseline executions with the memory-constrained Draco variant. The imposition of this memory limit triggered two profound behavioral shifts. First, the optimal tile size for the constrained Draco machine dropped from 400 to 125—exactly mirroring Beagle's optimal configuration. This shift is highly non-intuitive, as smaller tiles generate a larger volume of tasks, inherently increasing the overhead of the *scheduler's* task pool and memory traffic. Second, the constrained memory fundamentally altered Draco's response to logical threading. Specifically, restricting RAM significantly degraded Draco's optimal *makespan* when *hyperthreading* was disabled (worsening from 11.78 to 13.28 seconds). However, when *hyperthreading* was enabled, the constrained Draco maintained a stable *makespan* (13.21 seconds) virtually identical to its unconstrained baseline (13.29 seconds). This behavior indicates that the latency-hiding capability of *hyperthreading* effectively masks the increased memory access overhead induced by the RAM limitation. This resilience contrasts sharply with the Beagle machine, which exhibits a severe performance penalty under active *hyperthreading* (degrading from 14.07 to 17.60 seconds). Ultimately, these metrics definitively establish that RAM capacity acts as the primary governor of performance on these architectures, validating that Beagle's anomalous underperformance is fundamentally memory-bound.

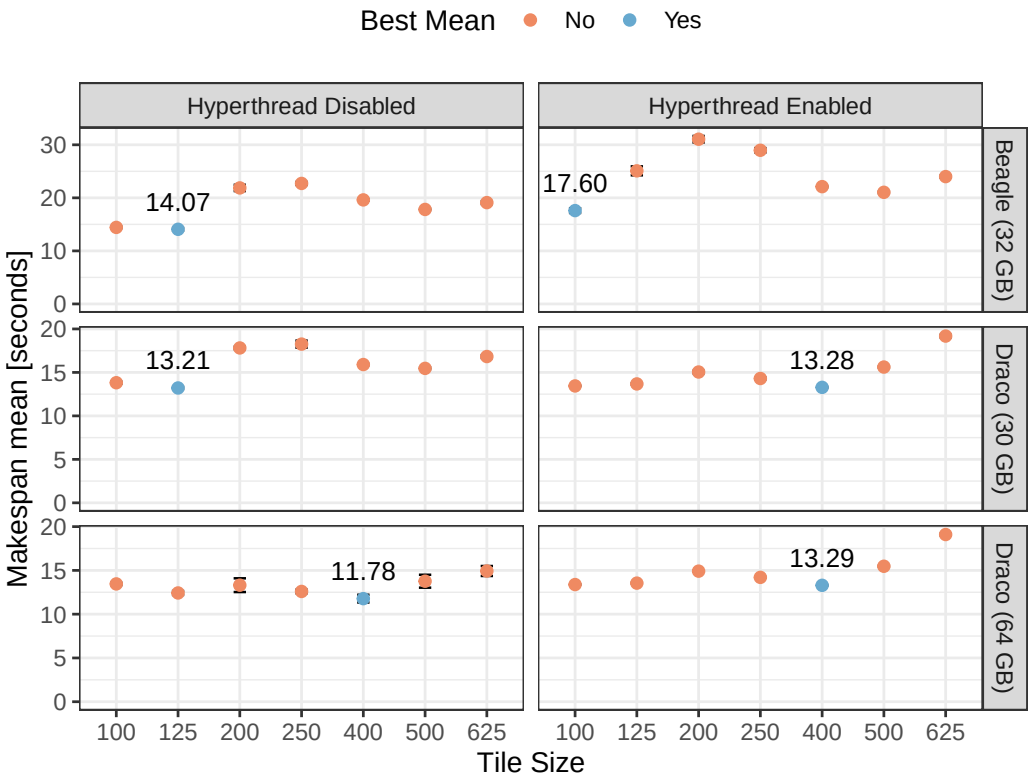


Figure 4.9 – Comparative makespan analysis of the Beagle machine against the Draco machine under standard and memory-limited configurations. The optimal mean makespan (in seconds) for each variant is annotated above the respective blue markers.

Finally, Figure 4.10 provides visual confirmation of how this hardware bottleneck manifests in the *scheduler*. Using a representative execution with a tile size of 200, the iteration charts demonstrate that restricting RAM drastically disperses the `DLARFB` tasks. The memory limit induces a massive temporal gap between the initiation of the first `DLARFB` task and the completion of the last, directly replicating the identical scheduling dispersion issue analyzed in subsection 4.2.6. This confirms that hardware-level memory starvation directly translates into the task-level scheduling delays visually identified by the proposed methodology.

In summary, mapping the task creation iteration to the vertical axis provides an analytical expressiveness that complements traditional profiling tools when diagnosing complex scheduling and hardware constraints. Nevertheless, while these subdivided views successfully isolate localized bottlenecks, maintaining distinct sub-rows per task type across massive trace datasets introduces new challenges regarding layout scalability and visual density. This challenge directly motivates the transition from partitioned rows to a continuous, aggregated representation of the computational volume—setting the foundations for the k-hills visualization technique detailed in the subsequent chapter.

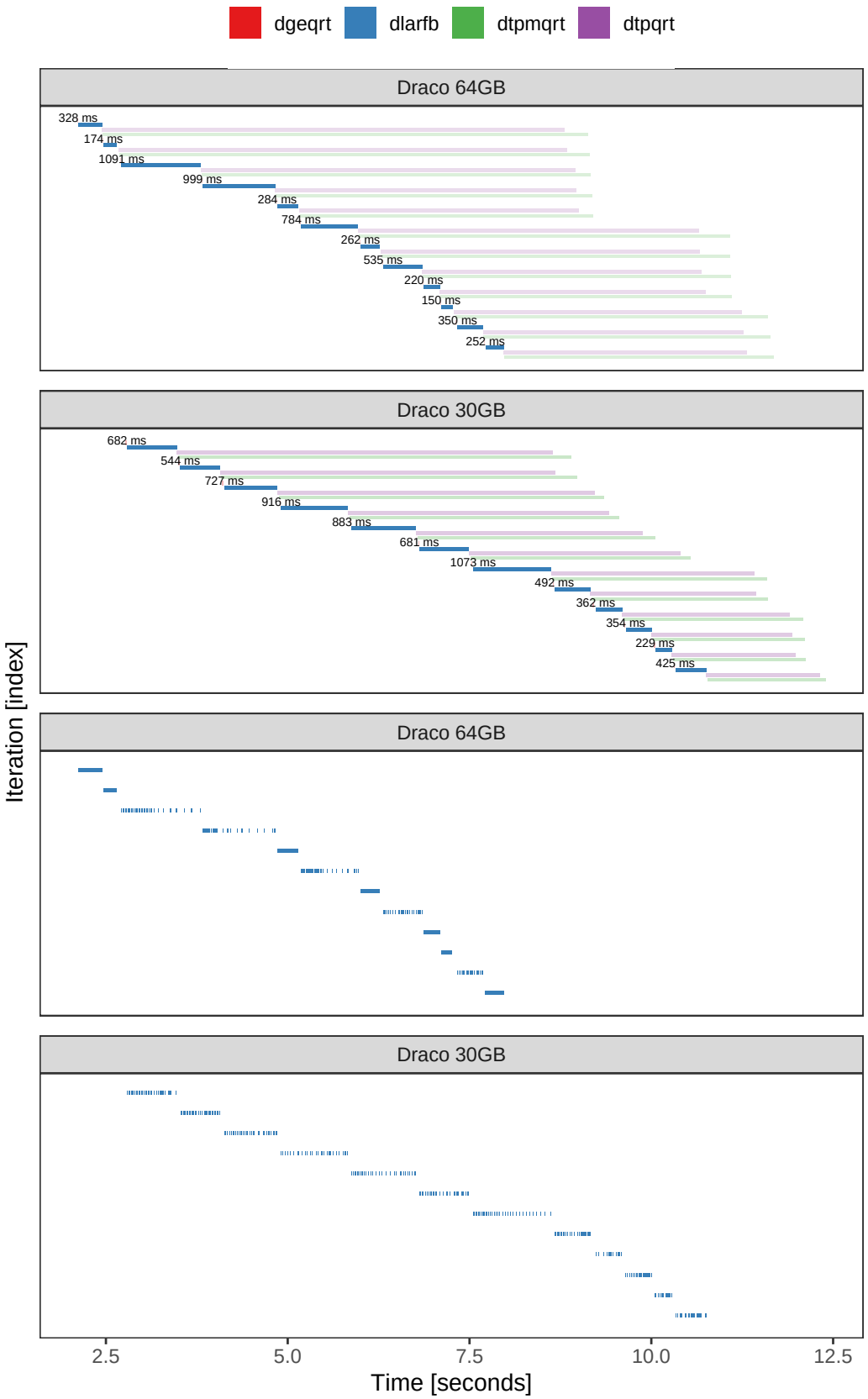


Figure 4.10 – Execution traces from the Draco machine (200 tile size, *hyperthreading* disabled, iterations 9 to 20) under constrained RAM. The top charts display bounding bars for `DLARFB` tasks annotated with their total temporal span, with other tasks faded. The bottom charts reveal the individual `DLARFB` task executions, demonstrating the dispersion and overlap induced by memory limitations.

5 THE K-HILLS VISUALIZATION: FORMULATION AND CASE STUDY VALIDATION

In the previous chapter[1], we demonstrated that traditional visual representations, such as Gantt charts, suffer from severe visual occlusion when processing the massive task volumes typical of high-performance applications. To mitigate this, we explored an initial discretization of the execution by splitting each iteration (k) on the vertical axis into distinct sub-lines for specific kernels. This approach successfully resolved spatial overlap and allowed us to track the temporal dispersion of individual tasks, such as `DLARFB`, revealing scheduling delays and hardware-level resource limitations.

However, the sub-line approach presents an intrinsic analytical limitation: while it exposes the *duration* and temporal *presence* of an iteration, it does not quantify the *volume* of computational power applied. Specifically, the previous visualization does not allow an analyst to determine the exact proportion of workers (processing cores) simultaneously dedicated to a specific iteration at any given microsecond. This lack of quantification hinders the diagnosis of subtle bottlenecks, such as dependency-induced starvation, where the system may appear active, but the critical path of the Directed Acyclic Graph (DAG) is actually stalled.

To address this gap, this chapter introduces the evolution of our visual methodology: **k-hills**. This new representation transforms temporal presence into a stacked quantification of computational effort, mapping precisely the fraction of the hardware dedicated to each task type throughout the algorithm's progression.

To validate the diagnostic efficacy of this evolution, we conducted a case study using tiled QR factorization on large-scale dense matrices. The evaluation directly contrasts two dynamic policies of the StarPU runtime—Work Stealing (WS) and Locality Work Stealing (LWS)—executed on a high-performance multicore architecture (the Sirius node, equipped with a 32-thread AMD Ryzen 9 3950X). The central objective is to demonstrate how visualizing computational volume replaces exhaustive manual DAG inspection, immediately exposing the exact moment and reason why one scheduling heuristic fails or outperforms another under the same hardware conditions.

[1] A preliminary version of this chapter was published in the proceedings of ERAD-RS 2026 (FUMACO; SCHNORR, 2026) and was awarded Best Graduate Paper (Melhor Trabalho de Pós-Graduação).

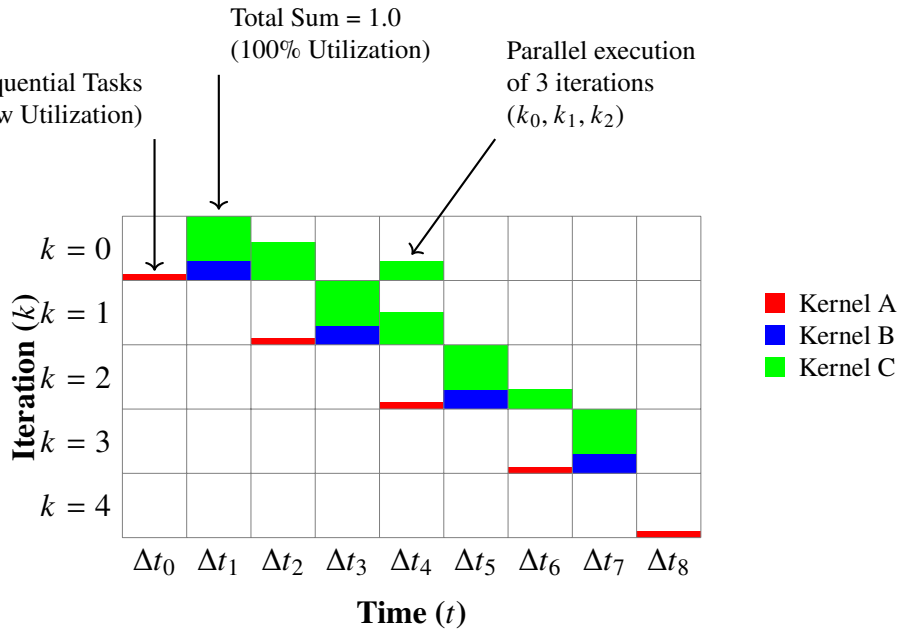


Figure 5.1 – Conceptual schematic of the k-hills visualization using generic kernels. The vertical distribution across iterations and the column sum demonstrate how system resources are partitioned over time.

5.1 The k-hills Visualization

To facilitate the interpretation and implementation of the proposed visual methodology, this section details the k-hills. This representation fundamentally shifts the analytical perspective from the individual lifespan of a task to the aggregated computational volume dedicated to algorithmic progression. By discretizing execution data into a structured 2D grid, the metric reconciles the temporal sequence of a Gantt chart with the structural depth of a Directed Acyclic Graph (DAG). Figure 5.1 illustrates the core geometric and mathematical principles of this metric through a conceptual schematic.

The generation of this chart relies on a three-step aggregation pipeline applied to the execution trace. First, the makespan is divided into uniform bins of width Δt, defined by the user. Choosing a large Δt yields a coarse mean that may obscure details, similar to what occurs in a standard histogram chart. Second, for each time bin Δt_i and each algorithmic iteration k_j, the runtime identifies the number of active workers. Third, execution time is normalized by the total available capacity within the bin, defined as $P \cdot \Delta t$ (where P is the total number of processing units). For a given bin, if a kernel k executes for a duration $t_{k,\text{bin}} \leq \Delta t$ using w_k workers, its proportional height is calculated as $h_k = (w_k \cdot t_{k,\text{bin}})/(P \cdot \Delta t)$. If a task spans across multiple bins, only the active time spent within that specific bin ($t_{k,\text{bin}}$) is counted. When multiple kernels within the same iteration

are active, their individual heights h_k are stacked vertically within the cell. The resulting grid uses the x-axis for continuous time and the y-axis for discrete iteration numbers (k). Unlike standard plots where markers overlap, this stacking logic ensures that every pixel represents a precise fraction of the machine's capacity.

The primary analytical power of this representation lies in the *Column Sum Rule*. In any vertical slice of the grid (representing a specific moment in time), the sum of the heights of all filled blocks across all iterations must satisfy the condition $\sum_{k=0}^{n} \text{height}(k, \Delta t) \leq 1.0$. A total sum of 1.0 indicates that 100% of the available processing resources are engaged in computation. Conversely, any value significantly less than 1.0 manifests as empty white space, explicitly revealing worker idleness or resource starvation.

As demonstrated in Figure 5.1, this metric exposes three critical execution signatures. Sequential bottlenecks, such as at Δt_0, occur when only the root task (Kernel A) is executing. Because this task acts as a sequential gatekeeper, it consumes only a small fraction of the machine, leaving the rest of the column empty and visually quantifying the cost of dependencies. Optimal local utilization, shown at Δt_1, occurs when an iteration has released enough internal parallelism for Kernels B and C to saturate the entire machine. Finally, multi-iteration concurrency, seen at Δt_4, reveals the scheduler's ability to "unroll" the algorithmic future. By assigning workers to tasks from $k = 0, 1$, and 2 simultaneously, the scheduler maintains a total column sum of 1.0, preventing the starvation that would occur if it strictly awaited the completion of earlier iterations. By observing these patterns, analysts can diagnose whether performance degradation stems from a lack of available tasks in the DAG or a failure of the scheduler to prioritize the critical path.

5.2 Case Study Setup

To empirically validate the diagnostic efficacy of the k-hills visualization, we designed a case study targeting a high-concurrency environment. The objective is to evaluate how effectively different scheduling heuristics manage the algorithmic critical path under heavy computational load, and to demonstrate how our proposed visualization exposes their underlying behaviors.

5.2.1 Hardware and Software Environment

The experiments were conducted on the Sirius node, hosted by the High-Performance Computing Center (PCAD) at the Federal University of Rio Grande do Sul (UFRGS). The node is equipped with an AMD Ryzen 9 3950X processor. This architecture features 16 physical cores with simultaneous multithreading (SMT) enabled, yielding 32 logical threads operating at a base clock of 3.5 GHz, supported by 64 GB of DDR4 memory. This high core density on a single socket provides a complex, highly concurrent environment where subtle scheduling decisions regarding task distribution and cache locality significantly impact the final makespan.

The software stack was strictly controlled to ensure reproducibility. The operating system is Debian GNU/Linux 12 (Kernel version matching glibc 2.36). The application and the underlying runtime, StarPU (version 1.4.7), were compiled using Clang 18.1.8. To handle the core numerical operations within the tasks, the system utilized OpenBLAS 0.3.27 and LAPACK 3.12.1. Execution traces, which are mandatory for generating our visualizations, were captured using the FxT 0.3.14 library, a lightweight tracing framework natively integrated with StarPU.

5.2.2 Workload and Parameter Tuning

The chosen workload is a tiled QR factorization applied to a massive dense matrix of $32,768 \times 32,768$ elements of double-precision floating-point numbers. The tiled QR algorithm was selected because its Directed Acyclic Graph (DAG) presents a rigorous test for schedulers: it features a strict sequential critical path (the panel factorizations, or `DGEQRT`) intertwined with massively parallel trailing matrix updates (`DTPMQRT`).

In task-based execution, the tile (or block) size is a critical configuration parameter. It dictates the granularity of the DAG. If tiles are too small, the runtime is overwhelmed by task management overhead; if they are too large, the system lacks enough independent tasks to keep all 32 threads busy, leading to hardware underutilization. To establish a rigorous baseline, we conducted an initial tuning phase evaluating block sizes ranging from 64 to 4096 across ten independent replications. The empirical results identified 1024×1024 as the optimal block size for this specific architecture. This configuration partitions the matrix into a 32×32 grid of tiles, providing an optimal balance between cache utilization and available parallelism. This block size was strictly maintained throughout all formal

experiments.

5.2.3 Target of Analysis: Scheduling Policies

The core of this case study is a direct performance confrontation between two dynamic scheduling policies provided by StarPU:

- **Work Stealing (WS):** This policy prioritizes global load balancing. Each worker possesses a local task queue. When a worker finishes its assigned tasks and becomes idle, it is allowed to scan the entire system and "steal" tasks from the back of the most heavily loaded queue. This strategy aims to keep all processors active, often at the expense of data locality, as stolen tasks may require transferring data across different cache hierarchies.
- **Locality Work Stealing (LWS):** This policy introduces architectural awareness. It builds a hierarchical tree of the machine's topology (e.g., sharing L3 caches or NUMA nodes). When a worker becomes idle, it first attempts to steal tasks from its closest topological neighbors. Only if the local neighborhood is entirely empty will the worker look further away. The primary goal is to minimize data movement and maximize cache hits.

For each scheduling configuration, we performed ten independent executions. We extracted the execution makespan to perform statistical significance testing and collected the rich FxT traces to populate the k-hills visualization and the starvation panels. By synchronizing the macroscopic execution time with the microscopic trace data, we provide a robust baseline to visually dissect exactly how and why one policy outperforms the other.

5.3 Analysis of Performance and the Starvation Symptom

To evaluate the macroscopic impact of the scheduling policies, we first analyze the total execution time, or makespan. While makespan is the ultimate metric of success in high-performance computing, it is merely a final symptom of thousands of microscopic scheduling decisions made throughout the execution.

Over the ten independent executions, the Work Stealing (WS) strategy yielded an average makespan of 467.93 seconds (Standard Deviation: 5.89 s). In contrast, the

Locality Work Stealing (LWS) strategy was consistently slower, averaging 478.75 seconds (Standard Deviation: 1.46 s). Because WS exhibited higher execution variability and our sample size was relatively small ($n = 10$ per group), we employed a non-parametric Wilcoxon rank-sum test to evaluate the difference. The test confirmed that the WS strategy is statistically significantly faster than LWS ($W = 1, p < .001$), a conclusion further corroborated by a Welch's two-sample t-test ($t(10.11) = -5.64, p < .001$).

Although a 10-second difference might seem marginal in absolute terms, it represents a persistent, structural degradation in efficiency. In the context of large-scale HPC deployments, where applications may run for days across thousands of nodes, a consistent degradation caused solely by the software scheduler translates into massive losses in both computational resources and energy.

The critical question, therefore, is not *who* won, but *why* the topology-aware LWS scheduler underperformed compared to the naive, global WS scheduler. To investigate this, we must look beyond the final makespan and examine the temporal utilization of the hardware.

In task-based runtimes, performance degradation is almost exclusively driven by resource starvation. Starvation occurs when the runtime fails to provide enough ready tasks to keep all available workers busy. To track this, we utilize StarVZ's `lackready` panel, which continuously monitors the ratio of ready tasks to available computing resources. As visually demonstrated in the lower lackready panel of Figure 5.2 in the upcoming section, whenever this ratio drops below 1.0 (meaning there are fewer ready tasks than the 32 available threads), the system is flagged as starved.

However, the defining anomaly occurs precisely in the middle of the execution. Under the LWS policy, the system suffers from a severe, sustained period of mid-execution starvation. Dozens of workers abruptly run out of ready tasks and idle, despite the fact that the matrix is still massive and the DAG contains thousands of pending operations. Conversely, the WS scheduler manages to navigate this exact same middle phase with almost zero starvation, maintaining a steady, uninterrupted flow of work to all 32 threads.

This symptom clearly indicates that the LWS scheduler is actively mismanaging the Directed Acyclic Graph. The tasks exist, but their dependencies are not being resolved fast enough to keep the machine fed. While the `lackready` panel successfully identifies *when* the machine is starving, it cannot explain *why*. To uncover the root cause—to see exactly which tasks LWS is prioritizing and which ones it is ignoring—we must apply the k-hills visualization to the execution traces.

5.4 Visual Diagnosis: Unveiling the Bottleneck

To uncover the root cause of the mid-execution performance penalty observed in the LWS policy, we apply the k-hills visualization to the execution traces. Figure 5.2 synchronizes our 2D visualization grid (top panels) directly with the `lackready` panel (bottom panels). Although the plot features numerical y-axis labels of 0.0 and 1.0 due to the underlying rendering library configuration, it functions strictly as a binary indicator representing the presence or absence of core starvation. A baseline value of 0.0 indicates perfect efficiency, where all workers are successfully fed with tasks (rendered as blank space); conversely, a peak value of 1.0 denotes that the system is operating under structural starvation, where the runtime fails to provide a sufficient volume of tasks (rendered as a colored block). By correlating these two views, we can visually pinpoint how the distribution of computing resources dictates the survival or collapse of the algorithmic critical path.

The Locality Trap (LWS Analysis): When observing the right side of Figure 5.2, the LWS grid reveals a stark structural anomaly shortly after the initial iterations. As highlighted by the red dashed box, the visual blocks representing worker allocation become vertically dense and heavily concentrated around iteration 10. The system dedicates almost its entire computational volume to resolving the massive trailing matrix updates (`DTPMQRT`) of previous iterations.

This visual signature captures the exact mechanism of StarPU's queue dynamics under hierarchical, locality-aware work-stealing. As workers process heavy local updates, newly released critical panel tasks (`DGEQRT`) remain trapped at the back of their queues. Because LWS strictly prioritizes topological proximity to minimize data movement, distant idle workers will search their own empty local neighborhoods rather than crossing the machine topology to rescue these globally critical tasks.

Visually, this manifests as a failure to unroll the DAG. The metric shows workers "anchored by locality" on old work. Crucially, looking at the starvation panel directly below this phase, we see severe, continuous spikes approaching 1.0 precisely between 250s and 350s. The mid-execution starvation is thus proven to be a direct consequence of ignoring the critical path in favor of strict topological proximity.

Critical Path Unrolling (WS Analysis): Conversely, the left side of Figure 5.2 exhibits a markedly different, highly efficient pattern for the WS scheduler. Instead of forming dense vertical blocks, the execution forms a faster-advancing diagonal, explicitly

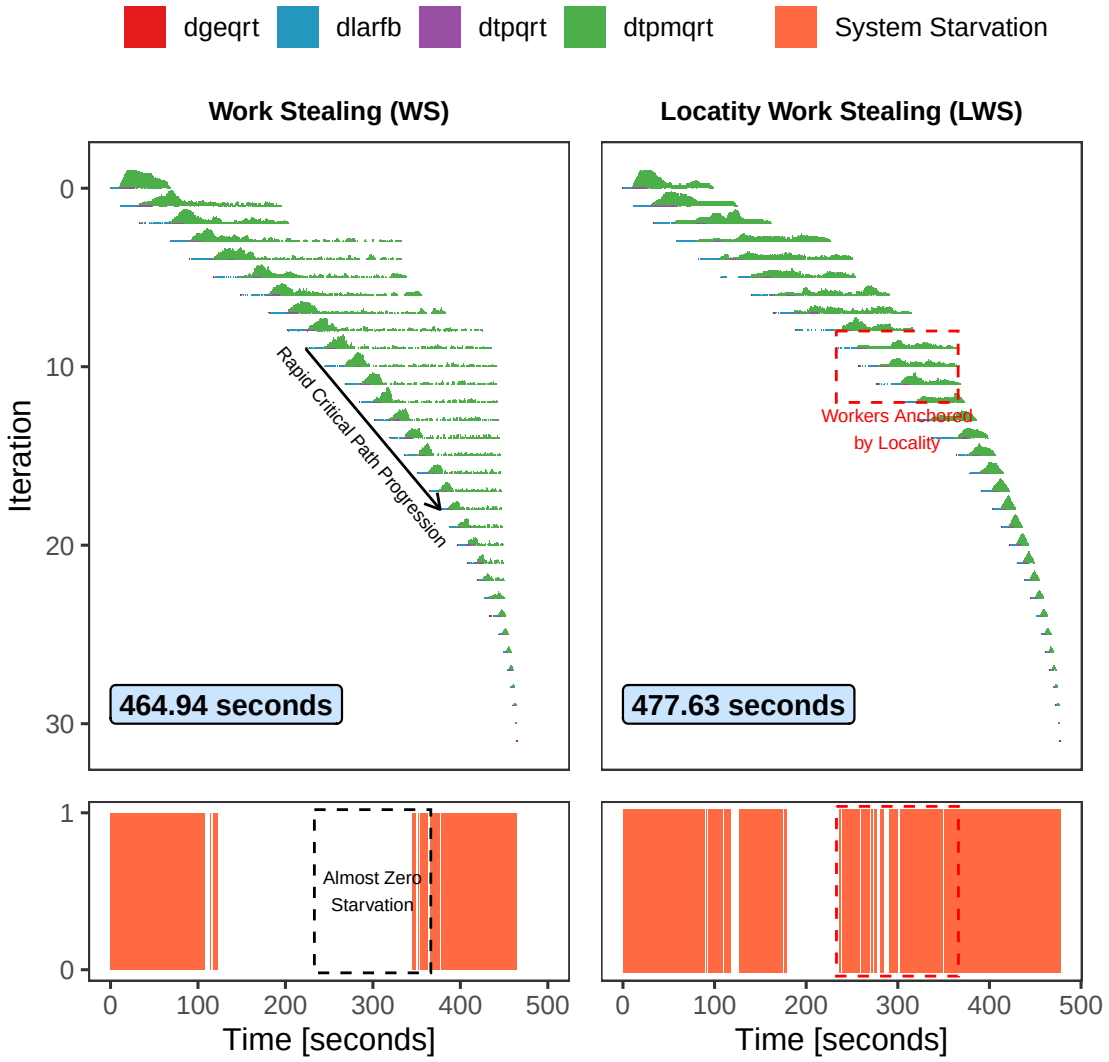


Figure 5.2 – Scheduling efficiency of WS vs. LWS. After iteration 10, LWS prioritizes cache locality, forming vertically dense execution blocks that delay the critical path and cause starvation (red). Conversely, WS aggressively steals critical panel tasks, forming a faster-advancing diagonal that unrolls the DAG, preventing worker idleness and significantly reducing makespan.

annotated as “Rapid Critical Path Progression”. The visualization clearly shows the system processing critical tasks for future iterations concurrently with updates from older iterations.

This visual shift is the direct result of WS’s aggressive global load balancing. Because WS ignores hardware topology, any globally idle worker is permitted to steal from the system’s most loaded queue. The moment a critical panel task is trapped, a starving worker immediately steals it, ignoring the penalty of cross-topology task migration to extract critical tasks early.

Our visualization captures this continuous global redistribution. By aggressively unlocking new branches of the dependency graph, WS guarantees a steady supply of future

tasks. This proactive unrolling perfectly reflects in the bottom panel: as highlighted by the black dashed box, the WS strategy yields a massive mid-execution phase of "Almost Zero Starvation" ($y \approx 0$). By continuously pulling the critical path forward, WS entirely prevents the idleness that penalizes LWS, resulting in the significantly reduced makespan.

Ultimately, this visual diagnosis demonstrates the power of the k-hills visualization. Rather than forcing analysts to hunt for a needle in a haystack—manually isolating the single, elusive task that blocks down-stream execution—the stacked 2D grid shifts the analytical focus toward macroscopic, task-type behavior. It translates abstract queue management algorithms into intuitive geometric patterns, shows exactly why prioritizing topological locality actively degrades performance in strictly constrained task-based algorithms. By successfully decoding the locality trap of the LWS policy and contrasting it with the proactive DAG unrolling of WS, the k-hills representation establishes itself as a robust, algorithm-aware diagnostic tool. It successfully translates complex, invisible queue dynamics into actionable visual signatures, empowering HPC developers to pinpoint and correct scheduling pathologies that would otherwise remain obscured within massive execution traces.

5.5 Diagnostic Proof of Concept: Exposing Implementation Anomalies

To demonstrate the portability, runtime-agnostic nature, and diagnostic sensitivity of the k-hills visualization, this section presents a comparative evaluation focusing on the visual tracking of task concurrency and scheduling behavior. We contrast the execution of the tiled QR factorization under three distinct scenarios: the same StarPU implementation analyzed in Chapter 5, a corrected version of this StarPU implementation under the same Work Stealing (`ws`) policy, and a baseline OpenMP execution. To enable a detailed, task-level analysis of the iteration dynamics, the evaluation focuses on the first three iterations of each scenario. While achieving highly comparable macroscopic execution times across these runs, their internal scheduling signatures reveal vastly different execution profiles. These structural nuances are captured in Figure 5.3, which serves as a visual guide to map task-level distributions to scheduling mechanics.

Directing our attention to the top panel of Figure 5.3, representing the StarPU implementation analyzed in Chapter 5, the k-hills visualization instantly exposes a structural anomaly at annotation (1). The `dlarfb` tasks manifest as a thin, horizontal blue line with virtually no vertical height, indicating that only a single worker is executing this task

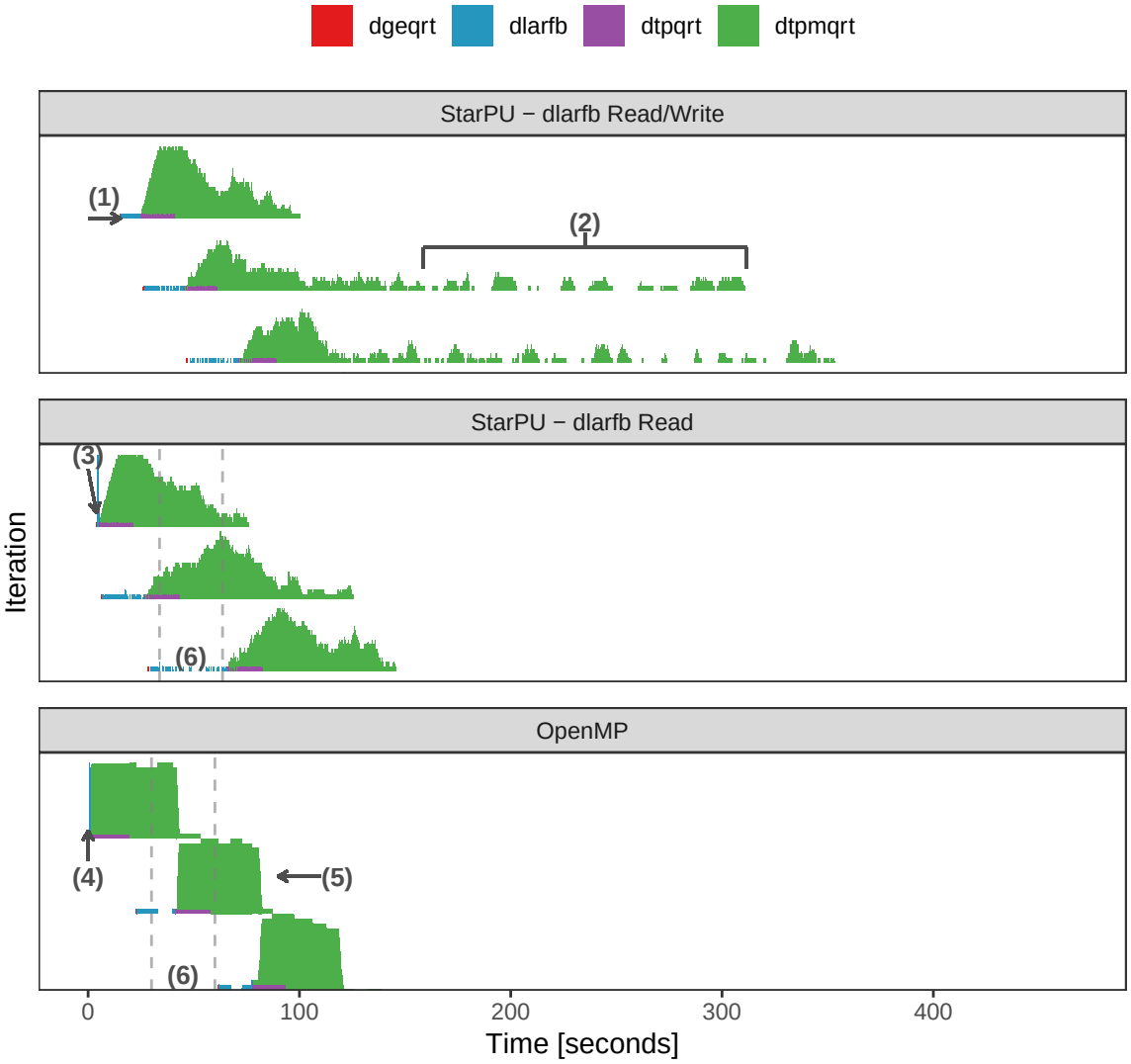


Figure 5.3 – Algorithmic progression signatures showing the first three iterations under different runtime configurations: (a) StarPU implementation from 5 featuring the sequential `dlarfb` dependency, (b) the corrected parallel `dlarfb` StarPU configuration, and (c) OpenMP baseline execution.

type at any given time bin. This evaluation served as a real-world debugging scenario for our visual methodology. The baseline tiled QR factorization code was inherited from prior development repositories (MILETTO; SCHNORR, 2019); through the diagnostic application of the k-hills visualization, an underlying constraint in the task dependency registration was successfully exposed. The sequential bottleneck was traced back to an inherited implementation oversight, where the task registration specified a Read-Write (RW) dependency on the matrix blocks instead of a Read-Only (R) dependency, artificially introducing sequential restrictions.

This localized bottleneck severely propagates through the task dependency graph, directly affecting the scheduling of downstream operations. As highlighted by annotation

(2) in the top panel, the sequential resolution of the `dlarfb` tasks forces the StarPU runtime to postpone a significant volume of trailing `dtpmqrt` updates. This deferral manifests as a sparse, prolonged tail of tasks dragging out to approximately 300 seconds, well past the dense computational core of the iteration.

Once this dependency constraint is corrected to Read-Only (R) in the middle panel of Figure 5.3, the physical concurrency of the execution is restored. This change is captured in the visualization at annotation (3) as a tall, vertical blue column at the beginning of the first iteration, indicating that multiple processing units are successfully executing the `dlarfb` tasks in parallel. This parallel execution is also clearly visible in the baseline OpenMP implementation shown in the bottom panel at annotation (4). Nonetheless, despite this structural change, both runtimes dynamically transition to sequential execution for these tasks in subsequent iterations, as shown by the flat, low blue lines in later stages, representing runtime-driven execution constraints. Crucially, the parallelization of this kernel in StarPU successfully eliminates the sparse task tail observed in the original implementation, resulting in a much more compact iteration boundary and a highly concentrated computational volume.

Beyond exposing localized bottlenecks, the visualization successfully decodes the fundamental scheduling philosophies of the different runtimes. In the bottom panel of Figure 5.3, the OpenMP execution signature at annotation (5) reveals a highly concentrated, greedy scheduling strategy. The OpenMP runtime aggressively concentrates its resources on completing the tasks of the oldest active iteration before fully committing workers to subsequent phases. This strict prioritization of the algorithmic critical path creates a highly structured, step-like diagonal progression of compact blocks, preventing the task dispersion observed in StarPU.

This difference in scheduling architecture is further exposed when analyzing the task execution granularity at annotation (6). In the corrected StarPU implementation, the `dlarfb` execution block is highly striped, characterized by alternating blue ticks and white gaps. This visual signature captures the dynamic interleaving of tasks, where workers under StarPU's localized, queue-per-core work-stealing model switch between different task types to maintain local load balancing. Conversely, the OpenMP execution in the bottom panel displays a solid, continuous blue block for the `dlarfb` tasks, with prolonged periods of complete task absence followed by continuous, uninterrupted execution phases. This visual pattern reflects a more centralized scheduling coordination, where threads continuously consume from a global task pool rather than localizing task execution.

Ultimately, this visual proof of concept demonstrates that the k-hills visualization is a highly sensitive diagnostic tool, successfully mapping subtle differences in data access descriptors and runtime scheduling paradigms into distinct geometric patterns.

6 CONCLUSION AND FUTURE WORK

This dissertation addressed a fundamental challenge in modern high-performance computing: the growing disconnect between the theoretical design of task-based parallel algorithms and the practical observability of their execution. As hardware architectures become increasingly complex and core counts continue to rise, runtime systems like OpenMP and StarPU dynamically manage thousands of dependencies and scheduling decisions per second. While this dynamic execution yields high performance, it renders traditional performance analysis tools—such as Gantt charts and statistical profiles—visually overwhelming at scale, often obscuring the structural algorithmic context.

To bridge this gap, this work proposed, developed, and validated a two-stage visual analytics methodology centered on algorithmic progression rather than chronological task lifespans.

6.1 Synthesis of Results and Achievement of Objectives

The primary objective of this research was to design an analysis methodology capable of exposing scheduler behavior and identifying performance bottlenecks in task-based linear algebra applications, specifically the tiled QR factorization. This objective was successfully achieved through an evolutionary, two-stage approach to execution visualization. In the first stage of this framework, we established a foundational iteration-centric paradigm that leverages the task submission iteration variable (k) on the vertical axis. By splitting execution traces into distinct sub-lines for specific kernels, this approach successfully mitigated the visual density issues that plague traditional timelines, demonstrating that embedding algorithmic depth directly into the visual representation allows analysts to track the temporal dispersion of individual tasks and observe macro-level scheduling delays.

Building upon these foundations, the second stage of our methodology introduced the k-hills visualization, which represents the core contribution of this work. By transforming the representation into a stacked 2D grid, we shifted the analytical focus to the volume of computational power applied. The metric enforces a strict column sum rule, mapping precisely what fraction of the hardware processing resources is dedicated to each task type and iteration at any given microsecond.

We validated the diagnostic efficacy of this methodology through a rigorous case study executing a dense $32,768 \times 32,768$ matrix on a 32-thread AMD Ryzen 9 3950X architecture. By directly contrasting the Work Stealing (WS) and Locality Work Stealing (LWS) scheduling policies of the StarPU runtime, our visualization methodology demonstrated its full diagnostic power. Standard macroscopic metrics indicated that LWS suffered a statistically significant performance penalty, but they could not explain its root cause. Our k-hills visualization eliminated the need for exhaustive manual inspection by allowing us to visually observe that LWS fell into a "locality trap"—anchoring workers to heavy, older matrix updates and starving the critical path. Conversely, the metric beautifully illustrated how WS aggressively unrolled the dependency graph to maintain maximum parallel throughput.

Ultimately, this dissertation succeeds in demonstrating that when execution traces are mapped proportionally to the algorithm's iteration space, complex queue management dynamics translate into intuitive geometric patterns. This empowers developers to pinpoint, explain, and correct scheduling pathologies that directly impact the application's final makespan.

6.2 Future Work

The methodology established in this dissertation opens several promising avenues for future research, both in expanding the visual model and applying it to more complex computational domains. First, regarding the expansion to sparse matrices and irregular DAGs, the current methodology targets dense tiled QR factorization, which yields a highly regular and predictable DAG structure. Future work should adapt the k-hills visualization to sparse linear algebra algorithms, such as sparse Cholesky or LU factorization, where task weights and dependency chains are highly irregular, making the visual tracking of computational volume even more critical for identifying structural bottlenecks.

Second, the framework should be extended to heterogeneous and GPU-accelerated systems. Modern HPC environments rely heavily on Graphics Processing Units (GPUs) alongside multicore CPUs. Expanding the proportional metric to account for different processing capacities is a vital next step. Since a single GPU worker consumes vastly different execution volumes compared to a CPU core, the visualization must represent normalized computational power, such as FLOPs per second, enabling the diagnosis of CPU-GPU load imbalance and data transfer starvation.

Third, the generalization to other algorithm classes represents a significant path forward. While this work utilized numerical linear algebra as the driving case study, the principle of mapping execution volume to algorithmic progression remains algorithm-agnostic. Future research could map this methodology to task-based graph processing algorithms, including PageRank and Breadth-First Search, or complex stencil computations, substituting the iteration variable (k) with graph traversal depth or time-step iterations.

Finally, integrating automated anomaly detection could significantly enhance the tool's utility. With the visual signatures of efficient and inefficient execution flows now formally defined, future iterations could leverage pattern recognition algorithms to automatically scan the proportional matrix grid, flagging periods of starvation or dependency locking in massive execution traces and recommending specific scheduler adjustments.

Acknowledgments

Experiments utilized the PCAD infrastructure (<http://gppd-hpc.inf.ufrgs.br>) at INF/UFRGS. This manuscript benefits from the use of AI tools during its preparation. Google Gemini assisted in structuring arguments and refining the narrative flow, while LanguageTool provided paraphrasing and grammatical revision.

REFERENCES

ADHIANTO, L. et al. Hpctoolkit: tools for performance analysis of optimized parallel programs. **Concurrency and Computation: Practice and Experience**, v. 22, n. 6, p. 685–701, 2010. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553>.

AGULLO, E. et al. Multifrontal qr factorization for multicore architectures over runtime systems. In: WOLF, F.; MOHR, B.; MEY, D. an (Ed.). **Euro-Par 2013 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 521–532. ISBN 978-3-642-40047-6.

AUGONNET, C. et al. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In: SIPS, H.; EPEMA, D.; LIN, H.-X. (Ed.). **Euro-Par 2009 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 863–874. ISBN 978-3-642-03869-3.

BHATELE, A.; BRINK, S.; GAMBLIN, T. Hatchet: pruning the overgrowth in parallel profiles. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2019. (SC '19). ISBN 9781450362290. Disponível em: <https://doi.org/10.1145/3295500.3356219>.

BHATELE, A. et al. Pipit: Scripting the analysis of parallel execution traces. 5 2024. Disponível em: <http://arxiv.org/abs/2306.11177>.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 46, n. 5, p. 720–748, set. 1999. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/324133.324234>.

BOARD, O. A. R. **OpenMP Application Program Interface**. 2008. Disponível em: <https://www.openmp.org/wp-content/uploads/spec30.pdf>.

COMPUTADORS, D. et al. Paraver: A tool to visualize and analyze parallel code. In: **Proceedings of the 18th World Occam and Transputer User Group Technical Conference (WoTUG-18)**. [S.l.: s.n.], 1995. v. 44.

COULOMB, K. et al. An open source tool chain for performance analysis. In: **5th Parallel Tools Workshop**. Dresden, Germany: [s.n.], 2011. Disponível em: <https://hal.science/hal-00707236>.

EICHENBERGER, A. E. et al. Ompt: An openmp tools application programming interface for performance analysis. In: RENDELL, A. P.; CHAPMAN, B. M.; MÜLLER, M. S. (Ed.). **OpenMP in the Era of Low Power Devices and Accelerators**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 171–185. ISBN 978-3-642-40698-0.

ERNSTSSON, A.; FRANKELL, E.; KESSLER, C. Interactive performance visualization and analysis of execution traces for pattern-based parallel programming. **International Journal of Parallel Programming**, Springer, v. 53, 10 2025. ISSN 15737640.

ESMAEILZADEH, H. et al. Dark silicon and the end of multicore scaling. In: **2011 38th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2011. p. 365–376.

FUMACO, A.; SCHNORR, L. Beyond the dag: Visualizing scheduling bottlenecks in task-based applications. In: **Anais da XXVI Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2026. p. 173–176. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/42729>.

GARCIA PINTO, V. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. **Concurrency and Computation: Practice and Experience**, v. 30, n. 18, p. e4472, 2018. E4472 cpe.4472. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472>.

GREGG, B. The flame graph. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 6, p. 48–57, maio 2016. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2909476>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 5th. ed. Waltham, MA: Morgan Kaufmann, 2011. ISBN 978-0123838728.

HUYNH, A. et al. Dagviz: A dag visualization tool for analyzing task-parallel program traces. 11 2015.

KURZAK, J. et al. Scheduling dense linear algebra operations on multicore processors. **Concurrency and Computation: Practice and Experience**, v. 22, n. 1, p. 15–44, 2010. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1467>.

MATOS, R.; SCHNORR, L. Intrusiveness and scalability of ompt-based tracing tools for task-based openmp applications. In: **Anais do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2025. p. 446–457. ISSN 0000-0000. Disponível em: <https://proceedings-sol.sbc.org.br/index.php/sscad/article/view/37919>.

MILETTO, M.; SCHNORR, L. Openmp and starpu abreast: the impact of runtime in task-based block qr factorization performance. In: **Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2019. p. 25–36. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sscad/article/view/8654>.

MILETTO, M. C. et al. Performance analysis of task-based multi-frontal sparse linear solvers: Structure matters. **Future Generation Computer Systems**, Elsevier B.V., v. 135, p. 409–425, 10 2022. ISSN 0167739X.

NESI, L. L. et al. StarVZ: Performance Analysis of Task-Based Parallel Applications. Working paper or preprint. 2020. Disponível em: <https://inria.hal.science/hal-02960848>.

REISSMANN, N.; MUDDUKRISHNA, A. Diagnosing highly-parallel openmp programs with aggregated grain graphs. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.]: Springer Verlag, 2018. v. 11014 LNCS, p. 106–119. ISBN 9783319969824. ISSN 16113349.

SAKIN, S. A. et al. Traveler: Navigating task parallel traces for performance analysis. **IEEE Transactions on Visualization and Computer Graphics**, IEEE Computer Society, v. 29, p. 788–797, 1 2023. ISSN 19410506. Good Related Work.I can describe the goals in the same way that they did in section 4.

SAKIN, S. A.; ISAACS, K. E. A literature-based visualization task taxonomy for gantt charts. In: **Proceedings - 2024 IEEE Visualization Conference - Short Papers, VIS 2024**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2024. p. 236–240. ISBN 9798350354850.

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. **Int. J. High Perform. Comput. Appl.**, Sage Publications, Inc., USA, v. 20, n. 2, p. 287–311, maio 2006. ISSN 1094-3420. Disponível em: <https://doi.org/10.1177/1094342006064482>.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. Upper Saddle River, NJ: Pearson, 2015. ISBN 978-0133591620.

WILLIAMS, K.; BIGELOW, A.; ISAACS, K. Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns. **IEEE Transactions on Visualization and Computer Graphics**, IEEE Computer Society, v. 26, p. 1118–1128, 1 2020. ISSN 19410506.

APPENDIX A — RESUMO ESTENDIDO

A transição em direção a arquiteturas multicore posicionou cada vez mais o paralelismo baseado em tarefas como um paradigma proeminente para a computação de alto desempenho (HPC). Embora sistemas de runtime agendem tarefas dinamicamente, suas heurísticas complexas de gerenciamento de filas tornam a análise de desempenho progressivamente difícil. As ferramentas tradicionais de *profiling*, fortemente dependentes de gráficos de Gantt baseados em linhas do tempo, muitas vezes tornam-se visualmente sobrecarregadas em larga escala e lutam para fornecer uma perspectiva orientada à aplicação, dificultando o diagnóstico de gargalos de desempenho. Esta dissertação propõe uma metodologia de análise visual em duas etapas para mitigar essa lacuna diagnóstica.

Na primeira etapa desse arcabouço, estabelece-se um paradigma centrado na iteração que reorienta os perfis de desempenho ao mapear as métricas de execução de tarefas para as iterações algorítmicas, utilizando a variável de iteração de submissão de tarefas (k) no eixo vertical. Ao dividir os rastros de execução em sublinhas distintas para *kernels* específicos, essa abordagem mitiga com sucesso os problemas de densidade visual que assolam as linhas do tempo tradicionais. Isso demonstra que a incorporação da profundidade algorítmica diretamente na representação visual permite que os analistas acompanhem a dispersão temporal de tarefas individuais e observem atrasos de agendamento em nível macro.

Construída sobre essas bases, a segunda etapa da metodologia introduz a visualização k-hills, que representa a contribuição central deste trabalho. Trata-se de uma técnica de visualização 2D condensada que muda o foco analítico do tempo de vida das tarefas individuais para o volume computacional agregado aplicado ao longo das iterações algorítmicas. Ao transformar a representação em uma grade 2D empilhada, o gráfico impõe uma regra estrita de soma de colunas que mapeia com precisão qual fração dos recursos de processamento do hardware está dedicada a cada tipo de tarefa e iteração em qualquer microssegundo. Esse mapeamento da fração de hardware dedicada a cada tipo de tarefa ao longo do tempo estabelece uma regra geométrica que expõe instantaneamente a utilização do hardware e as paradas induzidas por dependências.

Para fundamentar empiricamente a abordagem de visualização centrada em iterações, a metodologia de avaliação foi estruturada em três fases experimentais distintas, cada uma delas projetada para isolar e investigar cenários analíticos específicos utilizando

uma matriz densa de tamanho 10.000×10.000. A primeira fase experimental dedicou-se ao estabelecimento de uma linha de base de desempenho e à determinação do tamanho ideal de bloco (tile size) para cada arquitetura avaliada. Foram testados exaustivamente os tamanhos de bloco de 100, 125, 200, 250, 400, 500, 625, 1000 e 1250 em quatro plataformas de hardware distintas, alternando a execução entre cenários com hyperthreading ativado e desativado. Cada configuração única passou por 20 replicações independentes para assegurar a estabilidade estatística dos dados de tempo de execução coletados.

A segunda fase experimental concentrou-se em observar a resposta do escalonador a intervenções manuais no código da aplicação, limitando as avaliações de comportamento às máquinas Beagle e Draco. Foram introduzidas duas manipulações diretas no fluxo de agendamento: a atribuição de alta prioridade de execução explicitamente para as tarefas `DLARFB` e a imposição de uma barreira de sincronização estrita por meio da diretiva `#pragma omp taskwait` imediatamente após a submissão dessas mesmas tarefas. Essas intervenções foram testadas comparando o tamanho de bloco empiricamente ideal de cada máquina (sendo 400 para a Draco e 125 para a Beagle) contra um tamanho de bloco padrão e uniforme de 100, operando estritamente com o hyperthreading desativado e mantendo as 20 replicações por variante.

A terceira e última fase foi projetada para emular restrições físicas de hardware e isolar discrepâncias arquiteturais, limitando a máquina Draco a um teto máximo de 30 GB de memória RAM por meio da diretiva do Slurm `#SBATCH -mem=30G`. Essa restrição artificial reduziu a capacidade da Draco para um patamar operacional comparável aos limites físicos de memória da máquina Beagle, permitindo reexecutar a varredura completa de tamanhos de blocos da primeira fase para mapear e diagnosticar a degradação de desempenho induzida pelo barramento de memória.

A aplicação prática dos gráficos de iteração permitiu expor uma variação crítica no comportamento do escalonador OpenMP quando submetido ao threading lógico. Na arquitetura Cei, o isolamento visual das tarefas `DLARFB` revelou que a ativação do hyperthreading estende substancialmente o intervalo de tempo necessário para concluir essas operações dentro de uma mesma iteração. O detalhamento visual evidenciou que o hyperthreading faz com que essas tarefas se dispersem ao longo da linha do tempo, introduzindo lacunas significativas de ociosidade no processamento. Em contrapartida, a desativação do hyperthreading resultou em um bloco de execução visivelmente mais denso e coeso, indicando que o escalonador processa a fase de atualização de blocos de forma muito mais eficiente sob o isolamento de núcleos físicos.

Para validar se essa dispersão temporal configurava um gargalo estrutural, os testes com priorização explícita e barreiras de sincronização foram conduzidos utilizando também o tamanho de bloco reduzido de 100, cujo objetivo era inflar artificialmente a quantidade de iterações e amplificar o impacto cumulativo dos atrasos de agendamento. Os resultados quantitativos demonstraram que a priorização das tarefas `DLARFB` mitigou a dispersão observada, gerando reduções estatisticamente significativas no tempo de execução final das arquiteturas Draco e Hype. Nas plataformas Cei e Beagle, embora a variante padrão tenha superado nominalmente a versão com priorização, a aplicação do teste t independente resultou em valores-p de 0,263 e 0,5938, respectivamente, atestando que não houve degradação de desempenho estatisticamente significativa nesses ambientes. Dessa forma, a metodologia visual comprovou sua utilidade ao diagnosticar uma falha sutil de agendamento que pôde ser convertida em uma otimização direta de código de alta eficácia.

Em outra frente de validação, a arquitetura Beagle apresentou um platô de desempenho persistente em torno de 15 segundos em todas as configurações avaliadas, demonstrando um rendimento visivelmente inferior ao do nó Draco, apesar de ambas compartilharem especificações superficiais semelhantes. A máquina Beagle possui dois processadores Intel Xeon E5-2650 de 8 núcleos, enquanto a Draco é equipada com dois processadores Intel Xeon E5-2640 v2 de 8 núcleos. Apesar de a Beagle apresentar uma frequência de clock mais elevada em modo turbo, a Draco tira vantagem de uma microarquitetura de segunda geração (v2), que oferece maior taxa de instruções por ciclo (IPC) e melhor eficiência no subsistema de memória. Os gráficos de iteração revelaram de forma inequívoca que a Draco supera a Beagle de maneira sistemática a cada iteração individual do algoritmo. Essa evidência visual eliminou a suspeita de falhas no escalonador do runtime e isolou o problema a um déficit de hardware por iteração, direcionando a investigação para a capacidade da memória RAM, que é de 64 GB na Draco e de apenas 32 GB na Beagle.

A emulação de restrição de memória na Draco limitando seu espaço para 30 GB provocou duas mudanças de comportamento altamente reveladoras. Primeiramente, o tamanho de bloco ideal da máquina Draco reduziu drasticamente de 400 para 125, passando a espelhar perfeitamente a configuração ideal observada na máquina Beagle. Esse fenômeno é altamente contraintuitivo no escopo de HPC, dado que blocos menores fragmentam o problema em um número muito maior de tarefas, elevando o overhead de gerenciamento das filas do escalonador e intensificando o tráfego de dados na memória.

Em segundo lugar, a escassez de RAM alterou completamente a maneira como a Draco respondeu ao threading lógico. Sob a restrição de memória, o tempo de execução ideal da Draco sofreu uma forte degradação com o hyperthreading desativado, saltando de 11,78 para 13,28 segundos. No entanto, ao ativar o hyperthreading, a máquina Draco limitada conseguiu reter sua estabilidade temporal, registrando 13,21 segundos, um resultado virtualmente idêntico ao de sua linha de base irrestrita de 13,29 segundos. Esse comportamento comprova que os mecanismos de hardware voltados para a ocultação de latência presentes no hyperthreading foram capazes de mascarar o overhead de acesso gerado pelo gargalo de RAM. Essa resiliência contrasta diretamente com a máquina Beagle, que sofre uma penalidade severa de desempenho ao ativar o hyperthreading, piorando seu tempo de execução de 14,07 para 17,60 segundos.

Por fim, a confirmação visual desse diagnóstico foi obtida nos gráficos de iteração utilizando uma execução com tamanho de bloco de 200. Os perfis visuais comprovaram que a imposição do limite de RAM dispersou massivamente as tarefas `DLARFB` no tempo, criando um grande vão entre o início da primeira tarefa e o encerramento da última. Esse distanciamento mimetizou com precisão o mesmo padrão de dispersão identificado originalmente na análise de agendamento. O experimento consolidou a conclusão de que a inanição de memória em nível de hardware se reflete diretamente na eficiência do escalonador, traduzindo-se em atrasos visíveis no nível das tarefas e validando a capacidade diagnóstica da metodologia proposta.

A eficácia diagnóstica da metodologia k-hills foi expandida por meio de um estudo de caso focado na execução de uma fatoração QR em blocos densos com uma matriz de 32.768×32.768 em uma arquitetura de 32 threads. O estudo contrastou diretamente as políticas de agendamento dinâmico Work Stealing (WS) e Locality Work Stealing (LWS) do runtime StarPU. Embora as métricas macroscópicas padrão indicassem uma degradação de desempenho significativa para o LWS, elas não conseguiam explicar a sua causa raiz. A visualização k-hills decodificou essa anomalia com sucesso, eliminando a necessidade de uma inspeção manual exaustiva ao permitir a observação visual de que o LWS cai em uma armadilha de localidade, ancorando os trabalhadores em atualizações de matriz mais antigas e pesadas, o que deixa o caminho crítico em inanição. Por outro lado, o gráfico ilustrou como o WS desenrola ativamente o gráfico de dependências para manter a vazão paralela máxima e evitar a inanição no meio da execução.

Adicionalmente, as assinaturas visuais geradas pela ferramenta permitiram rastrear modificações de concorrência e comparar o comportamento de diferentes runtimes ao focar

nas três primeiras iterações do algoritmo. Na implementação original do StarPU analisada nesta dissertação, a visualização expôs uma anomalia estrutural: as tarefas `dlarfb` manifestavam-se como uma linha horizontal estreita e sem altura vertical, indicando uma execução estritamente sequencial provocada por um erro herdado no registro de dependência do tipo Read-Write (RW) onde apenas o acesso Read-Only (R) era necessário. Esse gargalo local propagava-se pelo gráfico de tarefas, gerando uma cauda esparsa e prolongada de atualizações de matriz `dtpmqrt` que se estendia até os 300 segundos. A correção da dependência para o modo correto eliminou essa cauda e restabeleceu a concorrência ideal, o que foi imediatamente traduzido pelo k-hills como uma coluna vertical de alta densidade na primeira iteração.

Ao confrontar esse comportamento com o do runtime OpenMP, a ferramenta também capturou distinções profundas nas filosofias de agendamento. Enquanto o OpenMP demonstrou um comportamento altamente concentrado e estruturado de forma gulosa, finalizando as tarefas da iteração mais antiga em blocos contínuos e sólidos antes de avançar, o modelo de filas por núcleo com roubo de trabalho do StarPU traduziu-se em assinaturas visivelmente listradas devido à alternância local de tarefas pelos trabalhadores para manter o balanceamento de carga.

Em suma, esta dissertação demonstra com sucesso que, quando os rastros de execução são mapeados proporcionalmente ao espaço de iteração do algoritmo, dinâmicas complexas de gerenciamento de filas se traduzem em padrões geométricos. Isso capacita os desenvolvedores a identificar, explicar e corrigir patologias de agendamento e limitações de hardware que impactam diretamente o tempo de execução final da aplicação, fornecendo uma ferramenta diagnóstica intuitiva e consciente do algoritmo.