Beyond the DAG: Visualizing Scheduling Bottlenecks in Task-Based Applications

Alisson dos Passos Fumaco[1], **Lucas Mello Schnorr**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{apfumaco,schnorr}@inf.ufrgs.br

Abstract. *Diagnosing starvation in task-based runtimes like StarPU typically requires tedious manual inspection of complex Directed Acyclic Graphs (DAGs). We propose a visual methodology that abstracts dependencies to evaluate critical path handling at a glance. Combining an enhanced k-iteration chart with StarVZ's lackready panel reveals scheduling bottlenecks without granular DAG analysis. We validate this approach by comparing Work Stealing (WS) and Locality Work Stealing (LWS) schedulers executing a tiled QR factorization. Visual evidence confirms WS achieves a statistically significant makespan improvement by sacrificing cache locality to unroll critical tasks, effectively preventing starvation and outperforming LWS.*

1. Introduction

Task-based programming models are essential for managing execution across heterogeneous high-performance computing (HPC) architectures. Runtimes like StarPU [Augonnet et al. 2009] abstract hardware complexity by representing applications as Directed Acyclic Graphs (DAGs) of tasks, delegating execution to dynamic schedulers. For instance, the default Work Stealing (WS) scheduler targets global load balancing X, whereas Locality Work Stealing (LWS) optimizes for architectural topology.

Performance debugging in these environments is notoriously difficult. Developers traditionally analyze post-mortem execution traces using Gantt charts in tools like ViTE [Coulomb et al. 2011] or explore task dependencies interactively via tools like Temanejo [Brinkmann et al. 2013]. StarVZ [Leandro Nesi et al. 2020] extends this analysis with a comprehensive suite of aggregated data panels. This suite includes a baseline k-iteration visualization to track algorithmic progress and the `panel_lackready`, which flags periods where the number of ready tasks falls below available computing resources.

While these tools effectively illustrate *when* workers starve, diagnosing *why* requires manually cross-referencing execution traces with the application's DAG to hunt down blocking tasks. As applications scale, the DAG devolves into an unreadable web of dependencies, making manual critical path analysis tedious and unscalable.

To bypass granular DAG inspection, we propose a visual analysis methodology that evaluates critical path scheduling at a glance. We introduce an enhanced, discretized variant of StarVZ's original k-iteration chart, coupled with `panel_lackready`, to highlight how efficiently a scheduler unrolls the critical path. We validate this approach by comparing StarPU's WS and LWS schedulers executing a tiled QR factorization.

Our methodology visually demonstrates that WS handles the critical path more effectively—achieving a statistically significant 10-second makespan reduction—without the researcher ever needing to manually identify blocking tasks within the DAG.

2. Experimental Setup

The experiments were conducted on the Sirius node from the Parque Computacional de Alto Desempenho (PCAD) at UFRGS. The system is equipped with an AMD Ryzen 9 3950X processor, featuring 16 cores and 32 threads operating at a base clock of 3.5 GHz, alongside 64 GB of DDR4 memory. The software environment consists of Debian GNU/Linux 12 with glibc 2.36. The application and StarPU runtime (version 1.4.7) were compiled using Clang 18.1.8. Linear algebra operations were supported by OpenBLAS 0.3.27 and LAPACK 3.12.1, while execution traces were generated using FxT 0.3.14.

For the workload, we executed a tiled QR factorization on a $32,768 \times 32,768$ matrix. An initial tuning phase evaluating tile sizes from 64 to 4096 across ten replications identified 1024 as the optimal block size. All formal experiments adopted this configuration with execution tracing enabled to generate the required performance visualizations.

To evaluate the performance difference between the Work Stealing (WS) and Locality Work Stealing (LWS) strategies, we measured the makespan across ten independent executions for each configuration. On average, the WS strategy yielded a lower makespan ($M = 467.93\,s$, $SD = 5.89$) compared to the LWS strategy ($M = 478.75\,s$, $SD = 1.46$). Notably, WS exhibited significantly higher execution variability than LWS. Due to these unequal variances and the small sample size (n=10 per group), a non-parametric Wilcoxon rank-sum test was employed. The test indicated that the WS strategy is statistically significantly faster than LWS ($W = 1$, $p < .001$). This conclusion was further corroborated by a Welch's two-sample t-test ($t(10.11) = -5.64$, $p < .001$).

3. Visual Analysis and Evaluation

To bypass manual DAG inspection, we introduce an enhanced visualization methodology that pairs a discretized k-iteration chart with StarVZ's starvation metrics (Figure 1). In our proposed chart, the x-axis discretizes execution time into uniform slices (e.g., 10 ms), while the y-axis represents the application's algorithmic progression (e.g., iterations in a tiled QR factorization). Within this grid, each cell contains a stacked bar whose total height indicates the fraction of total computing resources dedicated to that specific iteration during that time slice. The colors within these bars denote the proportion of specific task types executed (e.g., `dgeqrt` panel factorizations versus `dtpmqrt` trailing updates), revealing exactly how workers are allocated across the algorithmic depth. By temporally synchronizing this chart with the `lackready` panel below it—which plots the system starvation ratio—we can instantly correlate critical path delays with worker idleness.

Applying our visual methodology to the case study (Figure 1) reveals a stark divergence between the two schedulers around iteration 10. Initially, both display similar visual profiles: an abundance of trailing update tasks (`dtpmqrt`) provides sufficient parallelism to mask scheduling inefficiencies. However, as the un-factored matrix shrinks, the k-iteration chart exposes a severe bottleneck under LWS, manifesting as vertically dense execution blocks that stall algorithmic progression.

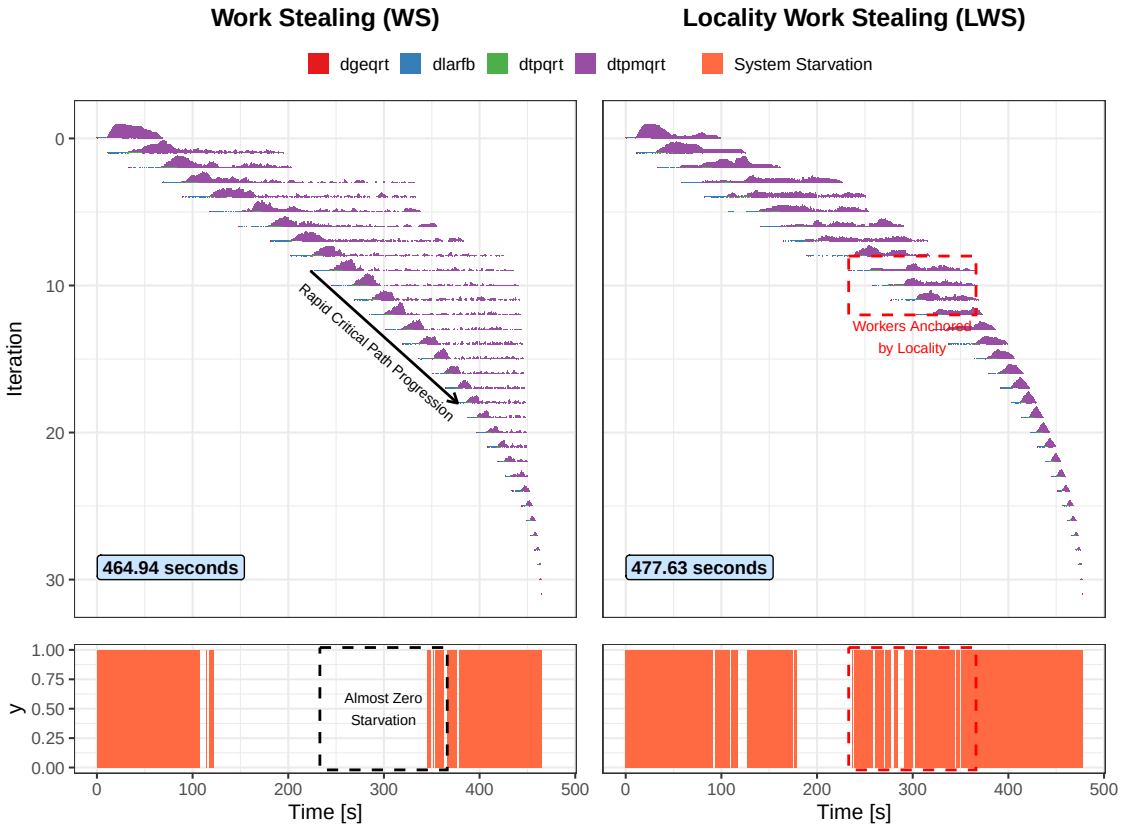

Figure 1. Scheduling efficiency of WS vs. LWS. After iteration 10, LWS prioritizes cache locality, forming vertically dense execution blocks that delay the critical path and cause starvation (red). Conversely, WS aggressively steals critical panel tasks, forming a faster-advancing diagonal that unrolls the DAG, preventing worker idleness and significantly reducing makespan.

This visual artifact perfectly captures StarPU's queue dynamics under hierarchical, locality-aware work-stealing. Workers push newly ready critical panel tasks (`dgeqrt`) to the back of their queues, behind older trailing updates. Because workers process their queues from the front, they become bogged down in older tasks. When other workers become idle, LWS directs them to steal from the back of neighboring queues to minimize data movement. However, if those immediate neighbors are also busy processing local updates, the newly unlocked critical tasks remain trapped. Distant idle workers are searching their own local neighborhoods rather than rescuing the globally most loaded worker. Our chart highlights this exact failure inside the red dashed rectangle labeled "Workers Anchored by Locality": the vertical blocks show workers grinding through local updates, while the synchronized `lackready` panel is visibly shaded in orange, demonstrating that the critical path is stalled.

Conversely, the WS chart exhibits a markedly different pattern: a faster-advancing leading diagonal composed of critical tasks (`dgeqrt`, `dlarfb`, and `dtpqrt`). This visual shift occurs because WS ignores topological proximity in favor of global load balancing. Any globally idle worker can steal from the back of the system's most loaded queue, immediately extracting the trapped critical tasks. By capturing this global redistribution, our chart visually illustrates how WS continuously unrolls the DAG, pulling the critical

path forward and drastically reducing the late-stage worker starvation that penalizes LWS.

The `lackready` panel further illustrates this impact. Mid-execution, WS maintains a steady work supply (unshaded regions), proving that globally stealing critical tasks successfully unrolls the DAG and sustains system parallelism. In contrast, because LWS traps workers in local updates, the stalled critical path blocks the release of new tasks. This manifests as substantial mid-execution resource idleness in the LWS plot, demonstrating that prioritizing topological locality over critical path progression actively starves the system and degrades the final makespan.

4. Conclusion

This paper introduces a scalable visual methodology to diagnose bottlenecks in task-based applications without tedious Directed Acyclic Graph (DAG) inspection. By coupling a discretized k-iteration chart with StarVZ's `lackready` metric, we visually abstract critical path scheduling. We validated this approach by comparing StarPU's Work Stealing (WS) and Locality Work Stealing (LWS) schedulers. Our analysis proves WS achieves a statistically significant 10-second makespan improvement by aggressively stealing critical panel tasks. This unrolls the DAG efficiently and prevents the severe mid-execution starvation that penalizes the locality-bound LWS. Ultimately, this approach empowers developers to identify scheduling bottlenecks at a glance. Future work will extend this methodology to irregular workloads and automate its integration into StarVZ.

Acknowledgments

Experiments utilized the PCAD infrastructure (`http://gppd-hpc.inf.ufrgs.br`) at INF/UFRGS. Per ERAD/RS guidelines, we declare the use of AI tools in preparing this manuscript. Google Gemini assisted in structuring arguments and refining the narrative flow, while LanguageTool was used for paraphrasing and grammatical revision. The authors rigorously reviewed all AI-generated suggestions and assume full responsibility for the paper's final content and scientific integrity.

References

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In Sips, H., Epema, D., and Lin, H.-X., editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brinkmann, S., Gracia, J., and Niethammer, C. (2013). Task debugging with temanejo. In Cheptsov, A., Brinkmann, S., Gracia, J., Resch, M. M., and Nagel, W. E., editors, *Tools for High Performance Computing 2012*, pages 13–21, Berlin, Heidelberg. Springer Berlin Heidelberg.

Coulomb, K., Degomme, A., Faverge, M., and Trahay, F. (2011). An open source tool chain for performance analysis. In *5th Parallel Tools Workshop*, Dresden, Germany.

Leandro Nesi, L., Garcia Pinto, V., Cogo Miletto, M., and Schnorr, L. M. (2020). StarVZ: Performance Analysis of Task-Based Parallel Applications. working paper or preprint.