Lock-Free Priority-Aware Work Stealing

Akshay Joshi (akshayjo@andrew.cmu.edu), Amogha Hegde (amoghah@andrew.cmu.edu)
Information Networking Institute
Carnegie Mellon University

***Abstract*—Efficient parallel scheduling is critical for modern multicore systems, especially in applications where task priorities influence computational quality or responsiveness. While classic work-stealing schedulers offer scalable, decentralized load balancing, they are typically oblivious to task priorities, leading to potential priority inversion and suboptimal execution in priority-sensitive workloads. In this work, we present a lock-free, priority-aware work-stealing scheduler that extends established lock-free deque designs to support efficient global prioritization without introducing centralized bottlenecks or excessive synchronization. Our algorithm employs multi-level, lock-free queues and atomic operations to ensure that higher-priority tasks are preferentially executed across all worker threads, minimizing priority inversion and maintaining high throughput even under dynamic and irregular workloads. We evaluate our implementation on a suite of benchmarks, demonstrating its effectiveness in balancing load, respecting task priorities, and scaling to many-core environments.**

***Index Terms*—lock-free, priority-aware, work stealing, CAS, priority inversion, load imbalance, steal overhead**

I. Introduction

Efficient load balancing is a cornerstone of high-performance parallel computing, particularly as modern applications increasingly rely on dynamic and irregular task structures. Among the various dynamic scheduling strategies, work-stealing has emerged as a leading approach for shared-memory architectures due to its decentralized design and scalability. In the classic work-stealing paradigm, each worker thread maintains a local double-ended queue (deque) of tasks, processing tasks from one end and allowing idle workers to steal tasks from the other. This strategy minimizes synchronization overhead and enables effective distribution of work across processors, making it the method of choice for a wide range of task-parallel applications.

However, conventional work-stealing schedulers are typically agnostic to task priorities, focusing solely on maximizing throughput and minimizing scheduling overhead. As a result, they may execute lower-priority tasks ahead of more critical ones, potentially increasing overall execution time or degrading the quality of results in priority-sensitive applications such as branch-and-bound algorithms, graph search, and soft real-time systems. While sequential priority scheduling is straightforward-often relying on priority queues-scalable parallel implementations face significant challenges due to the synchronization costs associated with shared concurrent priority queues. Existing solutions that attempt to integrate priorities into work-stealing either restrict prioritization to local queues or introduce global structures that compromise scalability.

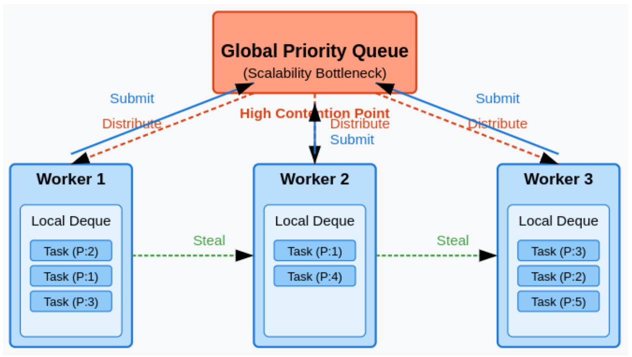

Fig. 1. Global Priority Queue

The need for priority-aware work-stealing is particularly acute in domains where the execution order of tasks directly impacts computational efficiency or result quality. For example, in search-based algorithms, prioritizing promising subproblems can dramatically reduce the search space. Similarly, in real-time and interactive systems, timely execution of high-priority tasks is essential for meeting responsiveness and quality-of-service requirements. Despite these benefits, most mainstream work-stealing frameworks do not natively support user-defined priorities, leading to suboptimal scheduling for such applications.

Moreover, the use of locks in traditional concurrent data structures can introduce contention and limit scalability, especially as the number of worker threads increases. Lock-free algorithms, leveraging atomic operations such as compare-and-swap (CAS), offer a promising alternative by enabling safe concurrent access without blocking, thereby reducing synchronization overhead and improving throughput. Recent research has demonstrated the feasibility and performance advantages of lock-free work-stealing deques, but integrating priority-awareness into such designs remains a challenging open problem.

In this project, we present Lock-Free Priority-Aware Work Stealing, an algorithm that combines the scalability of lock-free work-stealing with the ability to respect global task priorities. Our approach extends existing lock-free work-stealing deques to support efficient, decentralized priority management, ensuring that higher-priority tasks are executed preferentially across all worker threads. By employing multi-level queues and carefully designed atomic operations, our scheduler aims to minimizes priority inversion and synchronization overhead.

II. Background

Work-stealing is a dynamic scheduling technique widely adopted in parallel computing to efficiently balance workloads across multiple processors or threads. In this paradigm, each worker maintains a local double-ended queue (deque) of tasks. Workers execute tasks from one end of their deque and, upon becoming idle, attempt to "steal" tasks from the opposite end of another worker's deque, thereby redistributing work and minimizing idle time. The seminal algorithm by Blumofe and Leiserson, used in systems like Cilk [5], formalized this approach and demonstrated its scalability and efficiency for irregular parallel computations.

Work-stealing schedulers typically employ one of two policies when handling spawned tasks: work-first and help-first. Under the work-first policy, the worker executes the newly spawned task immediately, leaving the continuation available for stealing. Conversely, the help-first policy defers the spawned task, allowing other workers to assist by stealing it, while the original worker continues with the parent task. Each policy has distinct trade-offs in terms of stack usage, memory footprint, and scalability, with the work-first policy favoring deep, recursive parallelism and the help-first policy better suited for flat, wide task trees.

Traditional implementations of work-stealing deques relied on locks to ensure thread safety, but lock-based synchronization can introduce contention and degrade scalability, especially under high concurrency. To address this, lock-free data structures have been developed, leveraging atomic operations to ensure safe concurrent access without blocking. Lock-free deques, such as the Chase-Lev deque [6], have become the standard for efficient work-stealing, enabling high throughput and low latency even as the number of threads increases. These designs are particularly advantageous in multiprogramming environments, where locks can lead to preemption and further reduce parallel efficiency.

While classic work-stealing is effective for balancing load, it is inherently oblivious to task priorities. Many real-world applications, such as search algorithms, real-time systems, and interactive services, require that high-priority tasks be scheduled and executed preferentially. Integrating priority-awareness into parallel scheduling introduces significant challenges, primarily due to the complexity of maintaining global priority order without centralized, contention-prone data structures.

Recent research has explored decentralized approaches to priority-aware work-stealing. For example, Imam and Sarkar proposed a decentralized scheduler using multi-level queues [1], where each priority class has its own container and non-blocking operations are used to minimize synchronization overhead. Their design ensures that worker threads preferentially execute tasks from the highest available priority class, thereby reducing priority inversion and improving responsiveness for critical tasks.

III. Implementation

A. *Development Journey*

Our implementation is written in C, leveraging low-level atomic operations to construct lock-free deques. We use atomic CAS primitives to ensure thread-safe, non-blocking access to shared data structures, and explicit memory barriers (`__sync_synchronize()`) to enforce correct ordering of memory operations. The design is portable and does not target any specific hardware platform, but is optimized for multicore CPUs where each core can execute tasks independently. The approach is especially suited for modern shared-memory architectures, where hardware support for atomic instructions and efficient memory coherence protocols (such as MESI) allow for scalable, lock-free synchronization among worker threads.

Our implementation evolved through several key design iterations, each addressing specific limitations uncovered during development. The initial approach maintained a single lock-free deque per processor, storing tasks of all priorities together. While simple, this design led to frequent priority inversions and high overhead, as processors had to scan the entire deque to locate high-priority tasks, resulting in poor cache performance and inefficient scheduling.

To enforce stricter priority ordering and improve performance, we next introduced a global state to coordinate the current priority level across all processors. Although this centralized approach ensured that only the highest-priority tasks were executed at any time, it suffered from severe synchronization bottlenecks and race conditions. The need for frequent global coordination led to significant contention and poor scalability, ultimately making the system impractical for high-performance workloads.

Recognizing these drawbacks, we transitioned to a fully decentralized model in which each processor independently tracked its local priority level. Processors would exhaust all local and stealable tasks at their current priority before incrementing to the next. This eliminated the global bottleneck, reduced synchronization overhead, and allowed processors to make independent progress, greatly improving scalability and throughput. However, this approach introduced new challenges: processors could spend excessive time attempting unsuccessful steals, temporarily reducing overall efficiency.

To experiment the trade-off between strict priority adherence and execution efficiency, we limited each processor's steal attempts to a random subset-approximately $\sqrt{n}$ out of n total processors-rather than exhaustively searching all peers at each priority level. This approach reduces the time and overhead spent on unsuccessful steal attempts, allowing processors to more quickly advance to higher priorities and improving overall throughput, especially as the system scales. However, by not searching every possible victim, this method can increase the likelihood of priority inversion, where some lower-priority work may proceed before all higher-priority tasks are completed. In practice, this compromise significantly enhances performance under heavy and dynamic workloads, while still maintaining strong, though not absolute, priority guarantees.

B. *Various Scenarios in Work Stealing*

Below we explain some of the scenarios of work stealing that can occur.

*1) **Success Steal**:* Fig. 2 illustrates a scenario where, initially, Processor A has an empty deque at local priority 2, while Processor B's deque at the same priority holds three tasks (B1, B2, B3). When Processor A attempts to steal work, it successfully acquires the highest-indexed task (B3) from Processor B's deque. After the steal, Processor A's deque contains B3, and Processor B retains B1 and B2, demonstrating how idle processors dynamically balance load by stealing tasks from busier peers at the same priority level.

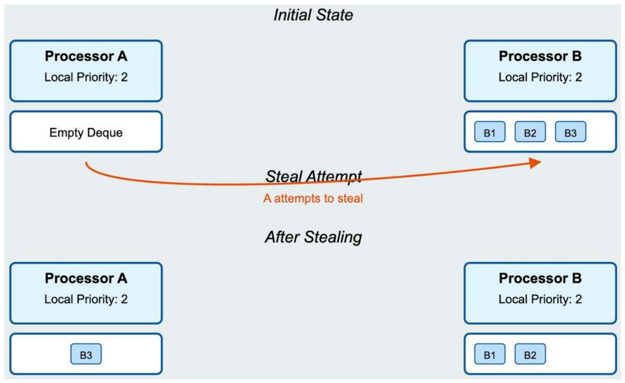

Fig. 2. Success Steal Scenario

*2) **Race Success Steal**:* Fig. 3 demonstrates a race condition among processors. Initially, only Processor C holds a task (C1) at priority 2, while Processors A and B have empty deques at the same priority. Both A and B simultaneously attempt to steal C1 from C, resulting in a race. After the race resolves, Processor A wins and acquires task C1, Processor B loses and increments its local priority, and Processor C's deque becomes empty. This scenario highlights how concurrent steal attempts are resolved and how unsuccessful thieves adjust their priority to continue searching for work, ensuring progress and load balancing in parallel scheduling systems.

*3) **Failed Steal**:* Fig. 4 illustrates a scenario where all processors at the same priority level have empty deques. Each processor attempts to steal work from the others, but since no tasks are available, all steal attempts fail. As a result, each processor increments its local priority, demonstrating the system's strategy for progressing to higher-priority work only after exhausting all stealing opportunities at the current priority level.

*4) **Priority Inversion**:* Fig. 5 depicts a priority inversion scenario. When both Processor A and Processor B attempt to steal a task from Processor C at priority 2, only A succeeds due to a successful CAS operation, while B fails and increments its local priority to 3. As a result, Processor B moves on to higher-priority work before all priority 2 tasks are completed, while A and C continue processing priority 2 tasks. This demonstrates how concurrent steal attempts can lead to temporary priority inversion, though the system eventually rebalances when all lower-priority tasks are finished.

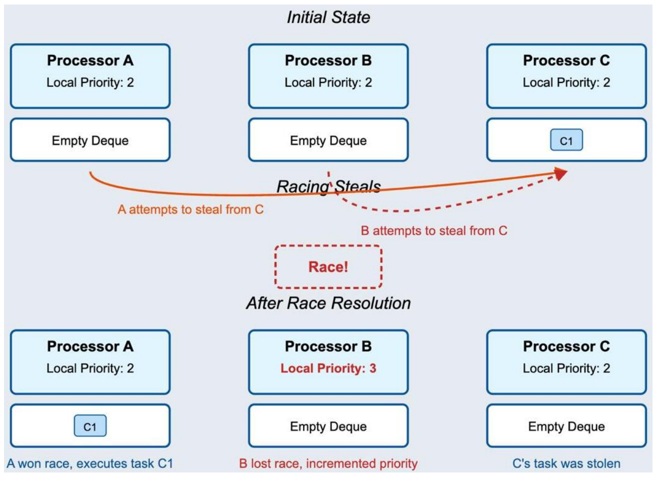

Fig. 3. Race Success Steal Scenario

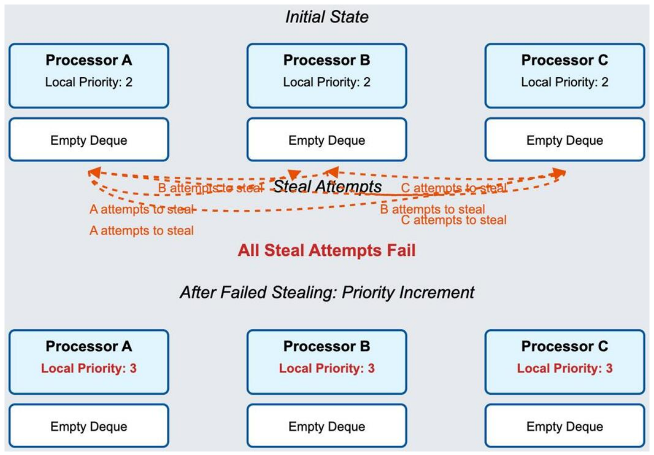

Fig. 4. Failed Steal Scenario

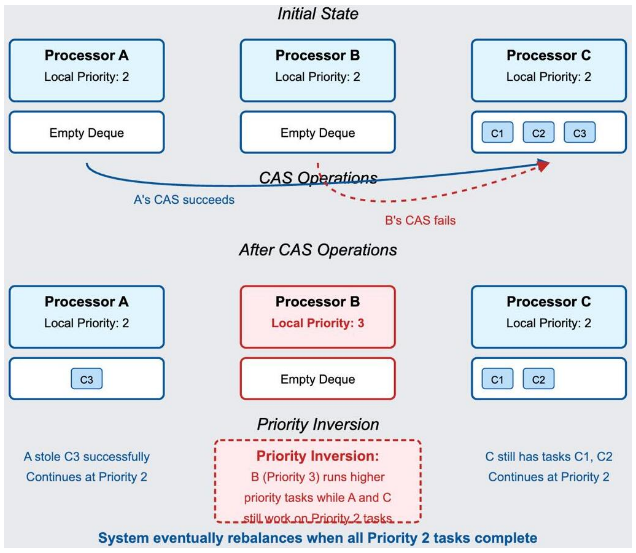

Fig. 5. Priority Inversion Scenario

C. *Overall Working*

Each processor in our system maintains a set of independent lock-free deques, with each deque dedicated to tasks of a fixed priority level; in our implementation, we use three priority levels. These deques are implemented using CAS operations, enabling processors to push and pop tasks without locks-local operations occur at one end of the deque, while remote processors steal from the opposite end, reducing contention. To ensure correct memory ordering and visibility of updates across all cores, we employ explicit memory barriers using `__sync_synchronize()` at key points in the push, pop, and steal operations. Each processor keeps track of the current priority level it is executing and initially works on tasks from its highest-priority deque. When a processor's local deque for the current priority becomes empty, it attempts to steal tasks of the same priority from other processors. If no tasks are found after probing all peers, the processor increments its local priority and begins working on the next lower level. CAS failures during stealing, which may occur due to contention, are tolerated by the system; a failed CAS does not guarantee that the deque is empty, only that another processor accessed it concurrently. This design accepts occasional priority inversions in favor of minimizing overhead from excessive retries, maintaining eventual consistency as processors repeatedly probe for work. Our implementation carefully balances the overhead of aggressive stealing with the need to enforce strong priority execution guarantees, and we evaluate this balance using metrics such as CAS failure rates, priority inversion occurrences, and average stealing effort.

IV. Results and Discussions

Our baseline comparison is with single lock-free deque per CPU that holds all tasks in it.

A. *Testcase Description*

`test_scenario_1`: In this file, every CPU has equal number of tasks and every deque per priority-level has equal number of tasks as well. Tasks within each deque take the same amount of time to execute.

`test_scenario_2`: In this file, every CPU has equal number of tasks and every deque for each priority-level has equal number of tasks as well. Tasks within each deque may take arbitrary amount of time to execute.

`test_scenario_3`: In this file, every CPU has equal number of tasks and but unequal division amongst the deques.

`test_scenario_4`: In this file, every CPU has unequal number of tasks.

`test_fib`: This file simulates a task queue system where multiple threads (peers) execute tasks with different priorities. Each processor gets equal number of tasks, but the tasks are distributed based on priority in a non-uniform way. Each task is a Fibonacci calculation with a random input. Each input is associated with a priority level (depending upon the defined range). The goal is to simulate uneven priority distribution and to ensure priority levels are respected.

`test_scaling_fib`: This file divides the tasks based on the number of CPUs. This is done to observe how the implementation performs when the number of CPUs increases. Additionally, changing the total number of tasks will help in determining the overhead of smaller number of tasks when CPUs scale.

`test_priority_inversion`: This file simulates a deviation from our work-stealing algorithm, where we do not check from all the CPUs before incrementing the priority. This is done to simulate a priority inversion scenario. This will perform better in terms of execution time as number of CPUs scale. However, there will be priority inversions, thus leading to lower-priority tasks being executed prior to higher-priority tasks in some cases.

B. *Analysis*

Fig. 6 represents the execution time taken for all tasks to complete on systems with 4, 8, and 12 cores. The benchmark used for this experiment is the scaling_fib test case. Upon analysis, we observe that the problem scales very well when moving from 4 to 8 cores. Specifically, the execution time decreases by slightly more than half. This indicates excellent parallel efficiency in this range. The primary reason for this behavior is that as we double the number of cores, there are still sufficient tasks available to keep all cores busy. In this regime, the overhead introduced by task stealing and synchronization is relatively small compared to the benefits gained by parallelizing work across more cores. The increase in parallelism outweighs the additional management cost associated with a larger number of processors. However, when moving from 8 cores to 12 cores, we no longer observe significant performance improvements. In fact, the execution time either remains roughly the same or in some cases slightly worsens. The primary reason for this is that the total number of tasks generated by the scaling_fib test case is not large enough to effectively utilize 12 cores. Once the main body of tasks has been distributed among 8 cores, the additional 4 cores in the 12-core system find fewer tasks to work on. As a result, these extra cores spend more time idle or competing for tasks through stealing mechanisms. Moreover, the overhead associated with synchronization and lock-free stealing operations becomes more significant as the number of cores increases without a proportional increase in available tasks. Every steal attempt introduces contention, cache invalidations, and failed CAS operations, all of which degrade performance. Instead of contributing meaningfully to throughput, the additional cores contribute primarily to overhead. Thus, beyond a certain core count, especially when the task granularity is small and the task count is fixed, the system shifts from a computation-bound phase to an overhead-bound phase.

Fig. 7 shows the execution time when running the same scaling_fib test but with fewer tasks than in Graph 1. Moving from 4 to 8 cores still gives a strong speedup of more than 50%, as the system can keep the cores busy without excessive overhead. However, when moving from 8 to 12 cores, performance worsens compared to the earlier case with

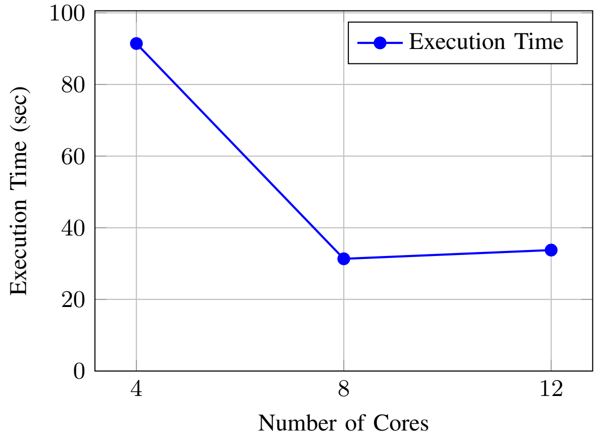

Fig. 6. Execution time vs Number of cores (no. of tasks = 2880)

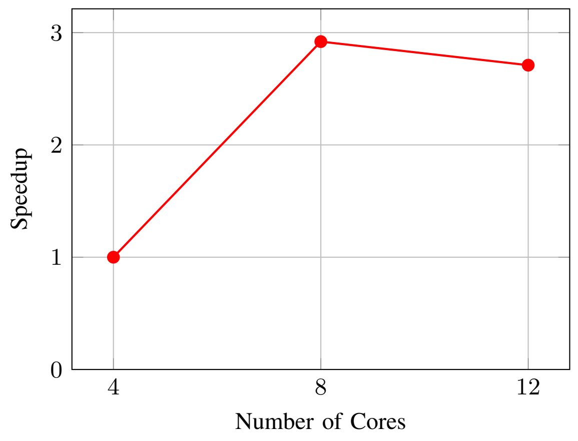

Fig. 8. Computational Speedup vs Number of Cores

more tasks. With fewer tasks, there is not enough work to fully utilize 12 cores, leading to increased idle time, higher synchronization overhead, and more costly task stealing. As a result, the execution time with 12 cores becomes worse than with 8 cores. Additionally, the improvement from 4 to 12 cores is only around 50% here, compared to about 66% with a larger number of tasks. This shows that when the task pool is small, adding more cores beyond a certain point leads to diminishing returns or even performance drops, due to the imbalance between available parallelism and management overhead.

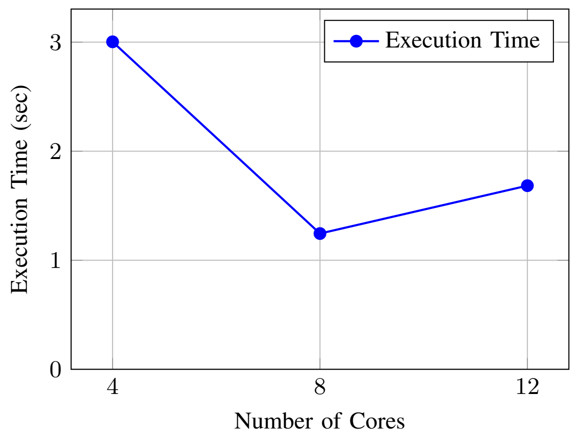

Fig. 7. Execution time vs Number of cores (no. of tasks = 144)

Fig. 9 represents the number of task steal attempts categorized by priority level in the test case `test_scenario_4`. For this experiment, we deliberately designed the task distribution to be highly skewed — that is, priority 0 tasks were unevenly assigned to different processors rather than being spread uniformly. Because of this skew, some processors quickly ran out of local priority 0 tasks while others still had many. As a result, processors that finished their own priority 0 tasks early attempted to steal tasks from others to continue processing the most important (priority 0) work first. This behavior led to a large number of steal attempts for priority 0 tasks, as seen clearly in the graph. The main insight from this experiment is that in workloads with significant imbalance in task distribution — especially where higher priority tasks are heavily skewed — priority-based stealing becomes very important. Without it, the system would prematurely move to executing lower-priority tasks simply because a core had no local high-priority work, leading to poor overall scheduling and delayed completion of important tasks. If the priority 0 (or critical) tasks are fairly balanced across cores, simpler schemes could suffice. However, if the workload is skewed, priority-aware stealing is crucial to maintaining good scheduling and avoiding priority inversions. This graph can therefore be used as a diagnostic tool for understanding whether priority scheduling overhead is justified based on the task distribution and workload characteristics.

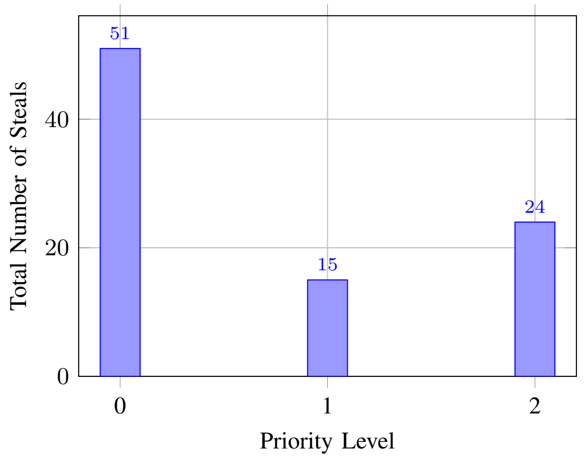

Fig. 9. Total number of steals per priority level in `test_scenario_4.c`

Fig. 10 represents the number of steals per core seen on a 4 core setup for testcase `test_scenario_4`. The motivation of this test is to see in case of load imbalances across cores how useful is the stealing overhead in comparison to staying idle

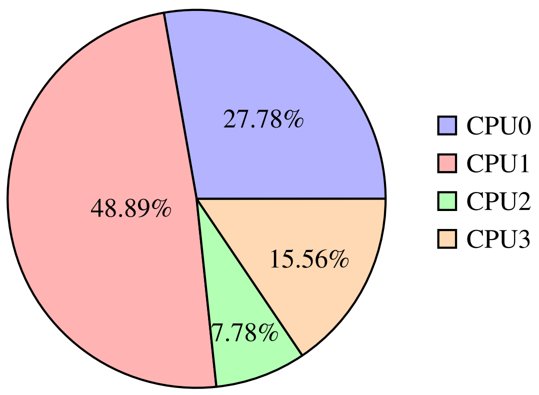

Fig. 10. Distribution of total number of steals per core in `test_scenario_4.c`

Fig. 11 is based on the priority_inversion test case and shows how priority inversions behave as the number of cores increases. In our original logic, there were no priority inversions because a processor would only move to the next priority after ensuring no other core had tasks of the current priority. However, to speed up execution, we modified the approach: once a core's priority queue is empty, it now checks only around $(\sqrt{n})$ other CPUs for stealing instead of all CPUs. As a result, the likelihood of missing tasks of the current priority (causing inversions) grows with the number of cores, roughly following a square factor. Thus, as the system scales up, the risk of priority inversions increases sharply. Overall, there is a trade-off between strict priority enforcement (which slows execution) and tolerating occasional inversions for faster task completion.

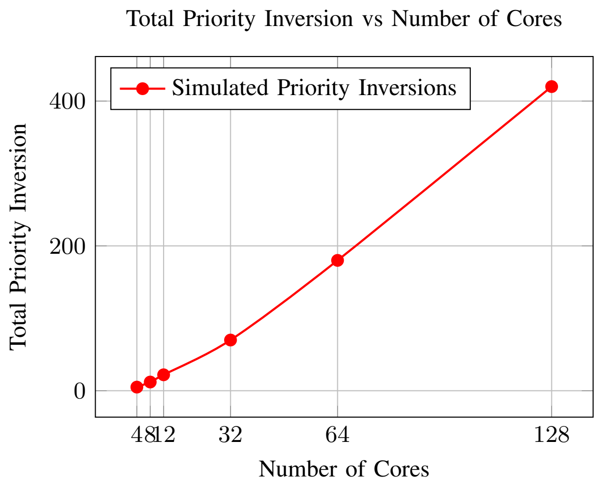

Fig. 11. Hypothetical increase in total priority inversion as cores increase (limited deque checking: $O(\sqrt{n})$)

Fig. 12 compares two different task scheduling approaches: a naïve lock-free deque system where all priority levels are combined into a single deque per processor, and our proposed priority-aware system that maintains a separate deque for each priority level. In the naïve approach, a processor continues to execute tasks from its deque without regard to their priority and only attempts to steal when its queue becomes empty. In contrast, our priority-based approach ensures that a processor only executes tasks from the current priority level and moves to a lower priority only after all tasks at the higher level are completed across all processors. Based on benchmarks across five test cases we developed, we observe that the priority-aware method is approximately 15% to 30% slower than the naïve one. This performance overhead is primarily due to the extra cost of stealing attempts even when lower-priority tasks are locally available, as the processor must still search for higher-priority tasks across all processors. Additionally, the priority checks introduce more synchronization and communication overhead compared to the simpler model. However, despite the slowdown, the priority-aware system ensures much stronger guarantees about task execution order, which is crucial for workloads where task priority is significant.

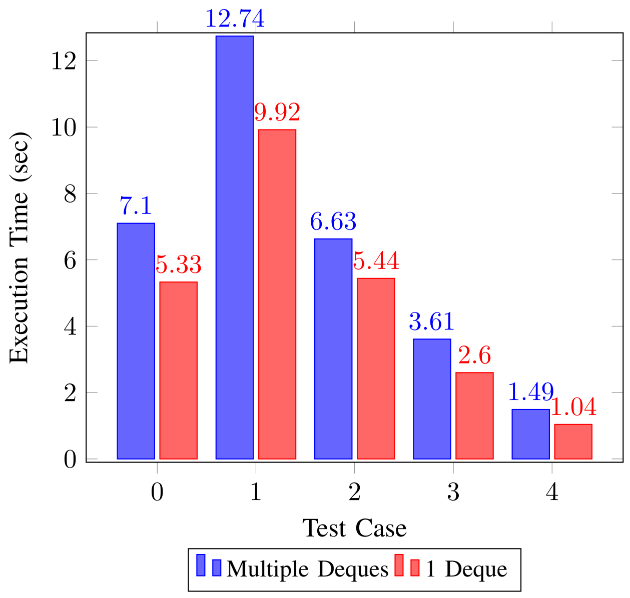

Fig. 12. Execution time comparison for 1 Deque vs Multiple Deques across test scenarios.

Fig. 13 represents the worst-case scenario for the time taken to complete all priority 0 tasks. When benchmarking against the non-priority-aware single lock-free deque solution, we observe that in the worst case, the single deque approach often results in the last task executed being a priority 0 task. In contrast, our priority-aware solution ensures that higher-priority tasks are completed first, preventing such delays. As a result, we observe more than a threefold improvement in the completion time for priority 0 tasks in our approach. This benefit becomes even more pronounced when the individual tasks are larger or when the distribution of tasks across processors is highly skewed toward priority 0. Overall, this graph highlights how strict priority enforcement can lead to significant performance gains in critical workload scenarios where task urgency matters.

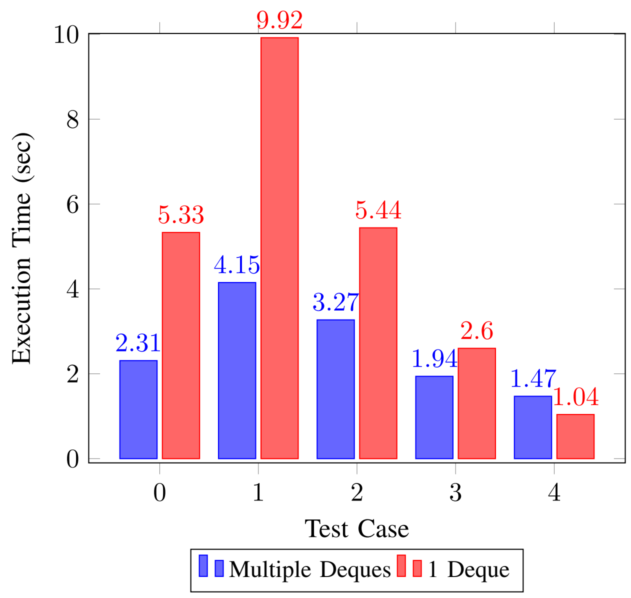

Fig. 13. Execution time comparison for 1 Deque vs Multiple Deques across test scenarios.

In our implementation, the major areas where the processors spend time, apart from executing, is during the situation of stealing tasks due to load imbalance w.r.t the high-priority tasks. A skewed priority distribution will lead to frequent steals, thus increasing the overhead. Additionally, some amount of time is also spent during the (`__sync_synchronize()`) operation to ensure values of top and bottom are visible to all processors in memory. We conclude that our target machine (multi-core CPU) was a good fit since GPUs better fit data-parallelism which was not the case in our project. Distribution of work done by each student: 50%-50%.

V. Acknowledgment

We would like to thank Prof. Todd C Mowry, and Prof. Brian P Railing, Computer Science Department, Carnegie Mellon University, to have provided us the opportunity and resources to work on this exciting project.

References

[1] Imam, Shams, and Vivek Sarkar. "Load balancing prioritized tasks via work-stealing." Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings 21. Springer Berlin Heidelberg, 2015.
[2] Prokopec, Aleksandar, Dmitry Petrashko, and Martin Odersky. "Efficient lock-free work-stealing iterators for data-parallel collections." 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2015.
[3] Prokopec, Aleksandar, Dmitry Petrashko, and Martin Odersky. "On lock-free work-stealing iterators for parallel data structures." (2014).
[4] Lockless Work Stealing Deque in C by Paran Lee and Sho Nakatani https://github.com/paranlee/lockless-work-stealing-deque-in-c
[5] Blumofe, Robert D., et al. "Cilk: An efficient multithreaded runtime system." ACM SigPlan Notices 30.8 (1995): 207-216.
[6] Chase, David, and Yossi Lev. "Dynamic circular work-stealing deque." Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. 2005.
[7] Compare-and-swap - https://en.wikipedia.org/wiki/Compare-and-swap
[8] `__sync` builtin functions - https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html