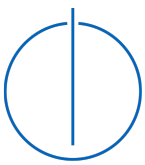
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low Latency Scheduling on Many-Core CPUs

Ismail Safa Toy



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low Latency Scheduling on Many-Core CPUs

Niedriglatenz-Scheduling auf Mehrkern-CPUs

Author:	Ismail Safa Toy
Supervisor:	Prof. Viktor Leis
Advisor:	Marcus Müller
Submission Date:	27.09.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 27.09.2024 Ismail Safa Toy

Acknowledgments

I am deeply grateful to my family, especially my mom and dad, for their unwavering support and encouragement throughout my academic journey. Your love and patience have been my greatest strength.

Finally, I would like to express my sincere gratitude to Prof. Viktor Leis for overseeing this thesis. My profound appreciation goes to Marcus Müller, whose guidance and expertise were invaluable throughout this voyage for wisdom.

Most importantly, to Jade.

Abstract

Work stealing has been proven efficient for multithreading and multi-programmed system, however it is not optimal for low latency scheduling. This thesis presents a new method for low-latency task scheduling for many-core CPUs, addressing the challenges posed by increasing core counts and NUMA memory hierarchies. We introduce a hybrid scheduling algorithm that combines the benefits of work stealing with a mailboxing technique, aiming to reduce scheduling latency and improve cache performance. Our implementation utilizes various concurrent data structures, such as the Arora, Blumofe, and Plaxton algorithm, the Chase-Lev algorithm, and split deques, to optimize task management. We also incorporated the morsels approach into our scheduler for better cache locality of the tasks.

The proposed hybrid scheduler is evaluated against classical work stealing and Intel's Threading Building Blocks (TBB) across various benchmarks, including recursive algorithms with varying levels of complexity and parallel for-loop tasks with different data distributions. Results demonstrate that our approach achieves comparable or superior performance to existing methods while significantly reducing scheduling latency, particularly for task startup and completion phases. Cache performance analysis and memory-fence and compare-and-swap operation counts further validate the efficiency of our mailboxing technique.

Key contributions include the design of a cache-aware partitioning scheme, a hybrid scheduling mode incorporating both work sharing and work stealing, and optimizations for concurrent deque implementations. These innovations address critical challenges in low-latency scheduling, such as non-deterministic stealing, inefficiencies with few tasks, and the high overhead of both time and resources for stealing attempts due to contention and synchronization costs.This thesis provides insights into the design of efficient schedulers for modern many-core architectures. It offers a foundation for future work to incorporate further research ideas.

Contents

1. Introduction

1.1. Motivation

In the evolution of computer architectures, several developmental phases have brought forth different problems and their respective solutions. From unified memory to non-uniform memory access (NUMA) bringing cache hierarchies, from single-core to many-core designs necessitating robust scheduling, modern CPUs have become more complex with each iteration. The latest trend is to increase computing power with more cores and more cache per core [11].Parallel programming is necessary for today's computer architectures to fully utilize their capabilities. In this context, many programming tools emerged: from libraries such as Java concurrency API [43] and Intel's OneTBB [32], to standalone programming languages like NVIDIA CUDA [41] and X10 [15], or language extensions such as OpenMP [19] and MPI [17] for automatic parallelization.

1.2. Problem Statement

At the heart of parallel programming lies the division of a program's execution into so-called tasks, which are dispatched to threads to be executed concurrently. The aforementioned tools give developers the ability to specify strategies for partitioning execution. Some also support different scheduling and dispatching schemes. However, traditional scheduling approaches face significant challenges as we move into the era of many-core CPUs with tens or hundreds of cores on a single chip. The increasing core count amplifies issues of load balancing, cache coherence, and memory access latency. In particular, low latency scheduling becomes crucial for a wide range of applications.

From a computational intensity perspective, logical division of the execution is relatively inexpensive. The real challenge lies in the orchestration of concurrent execution with minimal latency. Fully utilizing cores while maintaining low latency requires sophisticated load balancing, which in the literature is achieved either by Work Sharing, where a thread with work deliberately gives up some work, or Work Stealing, where threads without tasks steal from threads with work [5].

1.3. Significance of the Work

The need for low latency scheduling on many-core CPUs is becoming increasingly critical in various domains:

- **Real-time Systems:** In applications such as autonomous vehicles, industrial automation, and financial trading systems, even millisecond delays can have severe consequences [22].
- **Cloud Computing:** With the growing adoption of microservices architectures, efficient scheduling is crucial for maintaining responsiveness and managing resources effectively in large-scale distributed systems.
- **Scientific Computing:** Many scientific simulations and data analysis tasks require high throughput and low latency to process vast amounts of data in reasonable timeframes [22].

By addressing the challenges of low latency scheduling on many-core CPUs, this work aims to contribute to the broader goal of enabling more efficient, responsive, and scalable computing systems across these critical domains.

1.4. Work Stealing and Its Challenges

Work Stealing is a widely adopted and theoretically efficient scheduler for parallel computations. Its expected runtime with total instructions W, critical path length S on P processors is $\mathcal{O}(W/P + S)$, within a constant factor of optimal scheduling [9]. Despite its theoretical efficiency and widespread use in many programming tools, Work Stealing faces two significant challenges in practice:

1. **Work Distribution Latency:** The non-deterministic nature of work stealing can lead to suboptimal work distribution, especially at the start and end of computations. This non-determinism can increase latency as tasks may not be distributed to idle processors efficiently.
2. **Synchronization Overhead:** Work stealing typically relies on concurrent deques for task management, chosen for their $\mathcal{O}(1)$ time complexity. However, the synchronization mechanisms required for these deques, such as memory fences and compare-and-swap operations, introduce significant overhead. Studies have shown that even a single memory fence can account for up to 40% of the total runtime in some cases [18].

Additionally, stealing causes cache issues because of task migration, breaking cache locality [1]. These challenges highlight the need for improvements in work distribution strategies and more efficient synchronization mechanisms to enhance the practical performance of work-stealing schedulers.

1.5. Contributions

This work addresses some inefficiencies of Work Stealing in the context of many-core CPUs. Our key contributions are:

1. A cache-aware partitioning scheme: Inspired by "Morsel driven parallelism" [37], our partitioner uses morsels to enforce the cache locality of the whole range (part of the execution). Using Morsels dramatically improves cache performance without noticeable overhead.

2. A hybrid scheduling mode: We introduce a hybrid mode of scheduling incorporating both Work Sharing and Work Stealing. Mailboxing delivers work to threads before stealing is allowed, significantly improving task start times without causing substantial overhead in total runtime [1].

3. Deque optimizations: We experimented with various types of deque optimizations, such as Private, Half-Private, and Chase-Lev, which resulted in a scalable, cache-aware scheduler with provable efficiency and low latency.

1.6. Thesis Outline

The rest of this work is structured as follows: Chapter 2 discusses the background of this topic in more detail. Chapter 3 introduces the techniques used in our scheduler. Chapter 3 describes the concrete implementation. Chapter 4 evaluates the performance of our scheduler, and lastly, Chapter 5 discusses our work, its limitations, and future directions. We conclude with chapter 6.

2. Background and Related Work

The Background section of this thesis establishes the foundation for understanding a scheduler's principles, technologies, and practical considerations. It introduces key concepts that will build the later chapters.

2.1. Evolution of CPU Architectures

In the last decades, computer performance has become faster with each generation. Moore's Law, which predicts that the number of transistors in a dense integrated circuit doubles about every two years, while not a natural law, this industry trend has been observed from the 1970s to the present day, driving significant advancements in computing power and efficiency [10]. Initially, lithographical improvements enabled more transistors in a smaller package, directly translating to faster processors. For example, the manufacturing process improved from 180nm (Pentium III in 2000) to 32nm (Westmere in 2010). During this period, clock speeds increased from around 1 GHz to almost 4 GHz. It is also important to note that the effective increase in clock speed is higher due to Instructions per Cycle improvements [10].

Unfortunately, the lithographical improvements were not sustainable due to the physical properties and manufacturing difficulties in smaller manufacturing processes. As a result, the rate of improvement slowed down, and the focus shifted to increasing clock speeds and improving IPC. However, this approach faced also physical limitations, primarily due to power consumption and heat generation [11]. Around 2005, a paradigm shift occurred in CPU design: manufacturers began to increase the number of cores on a single chip instead of pushing for higher clock speeds. This marked the beginning of the multi-core era, with dual-core processors becoming mainstream and quad-core processors following shortly after [27].

CPUs have not only developed into multi-core systems but can also do simultaneous multithreading (SMT or Hyper-Threading). SMT enabled more computing power in the same die. Because a single core can execute two threads simultaneously, the CPU becomes more forgiving towards less optimal memory usage, as waiting for memory does not stall the entire core when a thread is expecting memory. This flexibility

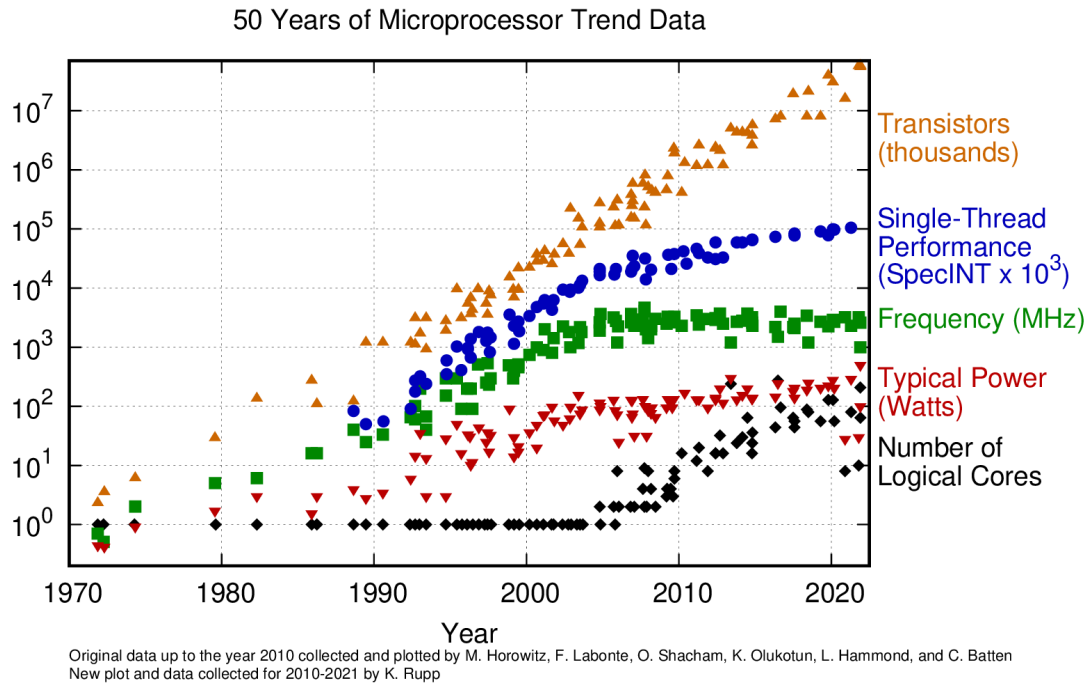


Figure 2.1.

allowed easier multiprogramming in the early 2000s [27, 11].

Moore's Law continued to be accurate, but the contributors to the increase in transistors changed. We observe a logarithmic development in clock speeds, power consumption, and performance per clock. This means improvements are in logarithmic relation, which is not appropriate for a linear increase in Moore's Law [10]. After a short stagnation, mostly due to a lack of competition, the number of cores is increasing steadily. Leading to the development of many-core CPUs, particularly for enterprise use cases. These processors boast significantly more cores than consumer processors, with some modern server CPUs featuring 64 or even 128 cores [3].

However, this shift introduced new challenges, often referred to as the "multi-core crisis" [48]. Adding more cores to a CPU often necessitated decreased clock speeds due to power and thermal constraints. Even today's many-core CPUs typically have clock speeds about 1-2 GHz lower than their consumer counterparts. Although many-core

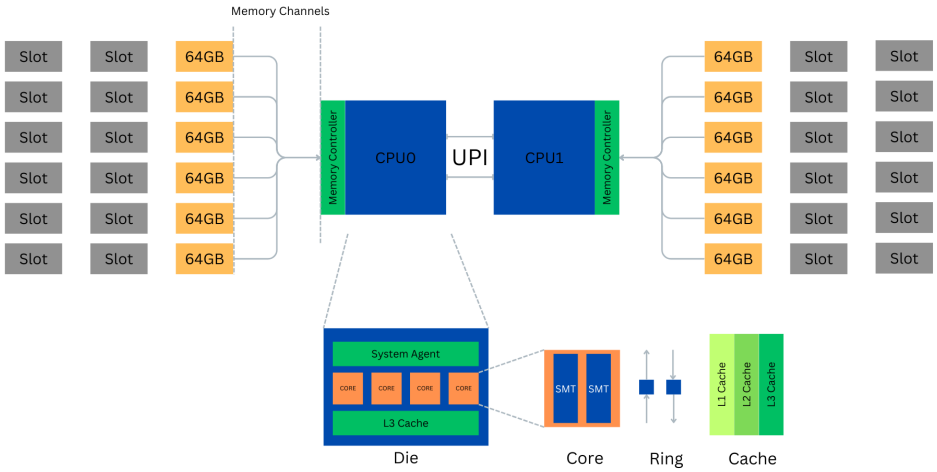


Figure 2.2.: Non Uniform Memory Access of a two-socket system visualized.

CPUs have more sophisticated memory arrangements, they often have a decade lead in terms of bandwidth, latency, and error correction. For example, the latest server CPUs often have 8 or 12-channel memory, Error Correction Code (ECC) memory support, and higher speed interconnects. These advantages allow us to saturate the cores [3].

The evolution towards multi-core and many-core architectures brought broader memory bandwidth and more cache per core. Still, memory outside of the processor needs to be accessed via the network connection; this operation stalls the core. Worse yet, the CPU operates considerably faster than the main memory it uses. That's why CPUs found themselves increasingly starved for data, also called Von Neumann Bottleneck. As the portion of memory fetching in the program execution increases, runtime becomes longer; this is especially true with multithreading. To minimize the time spent waiting, modern CPUs often feature complex cache hierarchies with multiple levels (l1, l2, l3), each with different sizes and access times [24]. It is important to note that cache access times have not improved in the last decade.

Cache performance has become crucial for overall system performance. Memory-intensive applications with suboptimal cache usage are slower, even with multithreading, than implementations with better cache utilization. It is because main memory access is prohibitively expensive compared to cache access, often by an order of magnitude or more [23, 24]. Data locality, or cache locality, must be cared for for optimal

performance. To mitigate this problem, modern processors employ techniques such as prefetching to minimize cache misses. Additionally, cache-oblivious algorithms and data structures can outperform those with better asymptotic runtime but worse cache performance [23].

Cache	Alder Lake P core	Alder Lake E core	Zen 4
Level 1 code	32 kB	64 kB	32 kB, latency 4 clocks
Level 1 data	48 kB, latency 5 clocks	32 kB, latency 3 clocks	32 kB, latency 4 clocks
Level 2	1280 kB, latency 15 clocks	2048 kB, latency 20 clocks	1 MB, latency 14 clocks
Level 3	4–30 MB, latency 65 clocks, shared	4–30 MB, shared	32–64 MB per 8 cores, latency 47 clocks

Table 2.1.: Cache sizes and latencies for Alder Lake (P and E cores) and Zen 4 architectures [24]

2.1.1. Cache Coherent Non-Uniform Memory Access (NUMA)

As core counts increased, traditional symmetric multiprocessing (SMP) architectures faced scalability challenges. This led to the development of Non-Uniform Memory Access (NUMA) architectures, where memory access time depends on the memory location relative to the processor [35]. NUMA introduces additional scheduling and memory management complexity, particularly in many-core systems. As shown above, optimally utilizing the cache is key to not incurring unnecessary memory fetch from the main memory, which can be around 150-200 cycles and even longer if the access is to remote memory [4]. Waiting for data is especially crucial in parallel programming, as waiting or blocking the execution impacts the whole execution time. Often, a simple inefficiency snowballs into a considerable chunk of wasted execution time.

NUMA enhances memory performance in two ways. Firstly, it reduces the memory latency for recently used data. More importantly, it reduces the amount of access to the main memory. Therefore, NUMA is beneficial for workloads with high memory locality of reference and low/no lock contention[4]. A hardware design challenge for NUMA architectures is to preserve coherence between caches of different NUMA clusters. A coherence problem stems from the tiered structure of the cache itself. Until Sandy Bridge-EP and Bulldozer architectures, the Last Level Cache (LLC) was not tiered or shared among all cores. As private caches (L1 and L2) per core emerged, the coherency between the caches needed to be maintained. The usual memory coherence protocol MESI can be used, though the communication for the Shared (S) state would

be too high on the system at this level. In a CPU with 64 cores with their respective caches, a read request can, in the worst case, trigger many cache snoops and their answers from the caches (cache lines) in state S. Modern cache architectures apply the MESIF protocol introducing Forwarding (F) state. One cache line is promoted to the F state, so it is the only cache that can respond to requests. The F state indicates "first among equals". The MESIF protocol reduces the traffic on the rings or interconnects [28].

Current trends are toward heterogeneous unified memory access (e.g., CPU + GPU) and coherently addressing remote memory. With more complex memory controllers, a CPU can directly address the memory of an accelerator. Modern designs often incorporate multiple levels of memory. AMD's infinity fabric allows tighter integration of shared memory among its devices (CPU and GPU). At the same time, NVIDIA's CUDA and Intel's OneAPI have been developed to provide better access to their respective platforms. The goal is to reduce memory-related bottlenecks. A CPU needs to request the execution data and communicate with the accelerator, and that accelerator needs to load the memory as well. In the hUMA architectures, a CPU can directly load the data related to the execution on the accelerator's memory.

2.1.2. Optimizations

Waiting for data is often the main cause of low performance. To mitigate this problem, modern CPUs employ a plethora of solutions to minimize this time. The most important optimization is prefetching. Prefetching loads a cache line (mostly 64 Bytes) into the cache before the region it was loaded from is requested. This is advantageous as it can be done parallel to another execution and is computationally cheap. Modern CPUs have built-in prefetching. Hardware prefetching is more efficient than explicit software prefetching in most cases [24].

Out-of-order execution (OOE) is another way to reduce the time spent waiting. It can be done by reordering the execution so that independent instructions can be executed while waiting for memory to be fetched. OOE is more suitable for unpredictable access patterns, as prefetching would miss the next access, thus causing overhead. For compute-bound bound workloads, OOE is more impactful, whereas prefetching shines at memory-bound workloads [23].

Branch Prediction and Speculative Execution are both probabilistic methods to select a branch in the execution that is likely to be the correct path in the conditional. The CPU continues to execute and/or prefetch the instructions and data in that path. If the executed path is wrong, then the executed pipelines are discarded. This is also

called a branch misprediction. The maximum wasted cycles are the length of a pipeline. A branch miss is more likely if the conditional is close to random. If the result of the conditional is distributed according to some distribution, the branch prediction is more likely to be correct [24].

2.1.3. Implications for Parallel Programming

In the face of stagnating clock speeds, IPC, and even memory access speeds, more cores might seem like the silver bullet needed for further development. Yet, since the introduction of the first dual-core CPU, it is evident that parallel programming is hard. Despite developing better parallel programming frameworks and hardware to run it on, many programs cannot leverage parallel programming to its full extent. The real improvement in computing power now lies less in waiting for faster hardware and more in writing cache-aware, scalable parallel programs that can efficiently utilize anywhere from 16 to thousands of cores [42, 18]. This evolution underscores the critical importance of effective scheduling algorithms, particularly those that can provide low latency in many-core environments. Schedulers must now contend with:

1. Increased core counts and potential for parallelism
2. Complex cache hierarchies and the need for cache-aware scheduling
3. NUMA effects and the importance of memory locality
4. The need to balance high throughput with low latency for diverse workload

In the context of this thesis, understanding these architectural trends is crucial for developing effective low-latency scheduling algorithms for many-core CPUs. The challenges posed by increased core counts, complex memory hierarchies, and diverse application requirements form the foundation for our research into advanced scheduling techniques.

2.2. Parallel Programming Models

In this section, we will give a brief overview of the current parallel programming models. Parallel programming is necessary to reap the benefits of hardware improvements. There are many models for parallel programming, but they can be roughly divided into two classes.

2.2.1. Process Interaction

In parallel programming terminology, some problems (e.g., embarrassingly parallel problems) do not require communication. These problems can be divided into self-contained subtasks. In that case, there is no need for process interaction. However, this case is rare in real-world tasks where many modes of dependencies need to be communicated between processes. Problems like weather prediction, heat diffusion, etc., cannot be calculated in parallel without process interaction. Communication can be done explicitly or implicitly.

Shared Memory

The shared memory model is the most common way to tackle communication on today's multi-core CPUs. It is fast and does not incur communication overhead with remote memory locations. Only considering the process communication without threads,

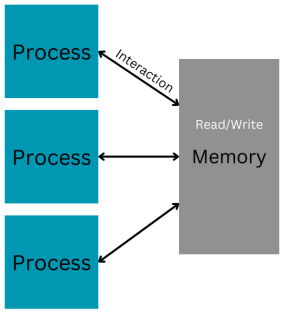


Figure 2.3.: Shared Memory Model without threads [36].

processes share a common memory address space that they read and write to asynchronously. In the case of concurrent access, synchronization needs to be implemented to avoid race conditions. Common interactions shape the execution, such as waiting, yielding, or notifying. Synchronization can be expensive if blocking other processes takes too long, limiting the gains of parallel programming. Considering threads as well, the situation looks similar with one difference: threads share the memory of a

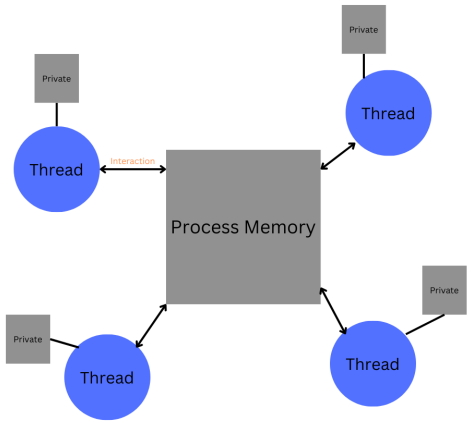


Figure 2.4.: Shared Memory Model with threads [36].

single process. This means the threads are not in a race condition with other threads of another process. Threads can be user-level or kernel-level. Changing the execution to another thread is cheaper than changing processes [21, 29].

One interesting area of parallel programming is the development of lock-free data structures. In this case, concurrent access to the data is not blocked, eliminating block and wait operations. Depending on the memory model, more care needs to be taken in order to guarantee the correctness of lock-free algorithms. The strong memory model (sequential consistency model) limits the possible reordering of the memory related to the execution by enforcing memory fences. Memory fence enforces the order of memory executions. On a high level, we can imagine four memory barriers:

- LoadLoad: prevents reordering of loads performed before and after the barrier.
- LoadStore: The processor is allowed to skip load operations if the coherence still applies afterward.
- StoreLoad: The latest stored value is visible to all other processors and loads after the barrier receives the correct value.
- StoreStore: prevents the reordering of the stores performed before the barrier with stores after the barrier.

Each restricts the reordering of the memory operations. In the strong memory model, all but StoreLoad reordering is prohibited, even though it acts like all of the barriers combined. This way, the compiler and the CPU cannot reorder the execution with more freedom, limiting the possibilities of more relaxed memory operations [44].

With the weak memory model, it is possible to use all of the reorderings. This way, the hardware is released of the burden of implicit acquisition and release of the memory (locking the memory essentially) [33, 44]. Another important concept is the atomic compare-and-swap (CAS) operation. In most modern hardware, CAS operation is implemented atomically on the hardware. Introduced in C++ 11, the compare_exchange_weak allows weak CAS operation, thus eliminating the need for a lock. Being able to use lock-free memory operations with cohesion guarantees lays the foundation of lock-free algorithms. In the context of parallel programming, not waiting for data is the key [23]. In this case, the main focus of the scheduler is to schedule threads in a nonblocking manner. Therefore, it is very important to use a weak memory model as well. Shared memory ensures low latency to memory access compared to message passing, but the scheduler needs to keep the NUMA principle in mind for cache locality.

Message Passing

In the message-passing model, processes have their own private memory, and communication is performed by sending and receiving messages. This model is well-suited for distributed systems, where processes may run on different machines or processors without shared memory. MPI (Message Passing Interface) is a common standard used in this model, which ensures scalability and fault tolerance in large-scale systems via explicit. Message Passing solves the limitation of scalability in Shared Memory models, though, while having more overhead in execution planning. The newest research is in asynchronous communication primitives primarily after MPI-3 using nonblocking collective operations in MPI-3, fault tolerance for exascale computing with User-Level Failure Mitigation (ULFM) proposal for MPI and automatic optimization of hybrid programming models (e.g., MPI + OMP) [17, 21].

Message-passing schedulers have to face the communication overhead of the model. Selecting the optimal nodes/clusters for communication is a challenge in itself. Because scheduling latency is not a big consideration for an exascale type of work, the focus is the saturation of computing power as well as data coherence (fault tolerance) [21].

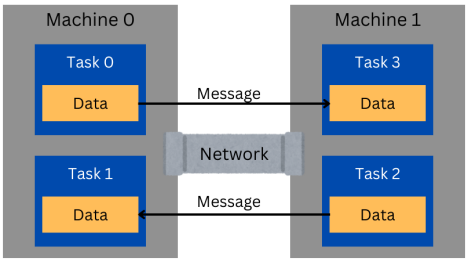


Figure 2.5.: Distributed Memory Model with message passing [36].

Partitioned Global Address Space (PGAS)

PGAS is a hybrid approach combining the shared memory aspect by logically partitioning the distributed memory. The whole memory is adressible to the process without the need for message passing. However, the internal implementation uses memory management similar to message passing. This often involves moving memory to caches of the clusters and/or propagating the updates in the system. Popular examples are Chapel [14], X10, and Unified Parallel C. Current research includes more efficient runtime environments with cheaper data movements, interoperability with existing frameworks such as MPI, and better scaling for exascale computing [21].

2.2.2. Problem Decomposition

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning. There are two basic ways to partition computational work among parallel tasks: data parallelism and task parallelism.

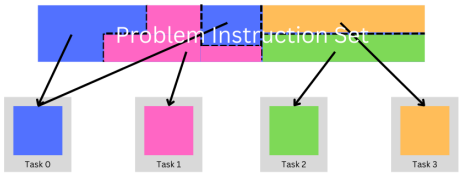


Figure 2.6.: Functional Decomposition Visualized [36]

Task Parallelism

Task parallelism is a parallel programming model that focuses on the concurrent execution of different tasks or processes. Unlike data parallelism, which simultaneously applies the same operation to multiple data elements, task parallelism emphasizes the distribution of distinct operations across multiple processing units.

In task parallelism, a problem is decomposed into separate tasks that can be executed concurrently. These tasks often have different behaviors and may require inter-task communication, making it well-suited for problems with diverse subtasks. Task graphs or Directed Acyclic Graphs (DAGs) are commonly used to represent task dependencies and guide execution order [21, 29]. Key considerations in task parallelism include effective load balancing, managing task granularity, and efficient scheduling. Work stealing algorithms are often employed to dynamically balance workload across processors. Popular frameworks supporting task parallelism include OpenMP tasks, Intel Threading Building Blocks (TBB), Cilk Plus (Deprecated), and Taskflow. These frameworks use both fork-join or task/thread pooling protocols. In the case of TBB and Taskflow, both can be used.

Task parallelism, particularly when implemented with work stealing, offers significant

advantages in load balancing and scalability. Work stealing dynamically redistributes tasks among threads, reducing idle time and improving overall system utilization. This approach is especially effective for irregular workloads or when task execution times are unpredictable. However, task parallelism faces several challenges. In work stealing, the act of stealing can introduce cache coherence issues, especially in NUMA architectures. When a task is stolen, its data may need to be transferred between NUMA nodes, incurring latency penalties. Moreover, frequent stealing can lead to cache thrashing, negatively impacting performance [21]. While necessary for maintaining data consistency, synchronization mechanisms like mutexes can become bottlenecks. Threads may experience significant waiting times due to lock contention, particularly in systems with high core counts. This can lead to decreased parallelism and increased latency, again showing the importance of lock-free algorithms. Task granularity presents another challenge. Too fine-grained tasks increase scheduling overhead, while too coarse-grained tasks limit parallelism. Achieving the right balance is crucial for optimal performance [21].

Lastly, the effectiveness of work stealing can be limited by task dependencies. Complex dependency graphs may restrict stealing opportunities, potentially leading to load imbalances and reduced parallel efficiency.

Data Parallelism

Data parallelism is a parallel programming model that focuses on distributing data across multiple processing units, which simultaneously perform the same operation on different data subsets. This approach is particularly effective for problems with regular data structures, such as arrays or matrices, where the same computation can be applied independently to many data elements [25, 29].

One of the key advantages of data parallelism is its scalability. As the data size increases, more processing units can be added to maintain or improve performance. This model also often leads to simpler program structures, as the same code is replicated across processing units, reducing the complexity of parallel programming [21].

However, data parallelism faces challenges in load balancing, especially when data is non-uniformly distributed or when computations on different data elements take varying amounts of time. Communication overhead can also become a bottleneck, particularly when data needs to be exchanged between processing units or when dealing with distributed memory systems [20]. Modern implementations of data parallelism often utilize SIMD (Single Instruction, Multiple Data) instructions available in many

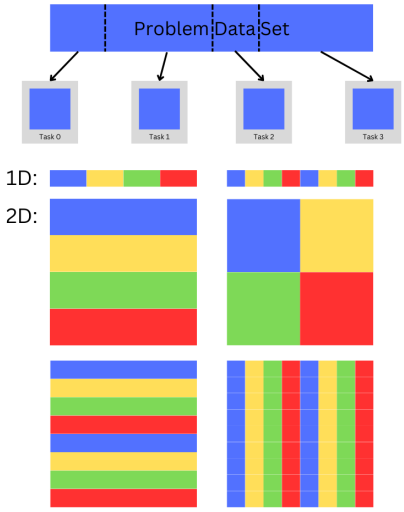


Figure 2.7.: Domain Decomposition Visualized [36]

processors, allowing for efficient vectorization of computations. GPU computing has also embraced data parallelism, offering massive parallelism for suitable problems [21, 29]. Recent research in data parallelism focuses on improving its applicability to irregular problems, enhancing load-balancing techniques, and integrating it with other parallel programming models for hybrid approaches [21].

More often than not, a combination of the two is used to maximize parallelism. This classification is not strict. Some problems can benefit from both task and data parallelism, namely matrix multiplications, image operations, etc. For high-performance computing, hybrid approaches are becoming more popular [21].

2.2.3. Low Latency Scheduling

Defining Scheduling Latency

Scheduling latency refers to the time delay between when a task becomes ready for execution and when it actually begins executing on a processor. Formally, we can define scheduling latency as:

$$L_s = T_e - T_r \tag{2.1}$$

where L_s is the scheduling latency, T_e is the time when the task starts executing, and T_r is the time when the task became ready for execution [47].

Scheduling latency encompasses several components [31]:

1. Decision time: The time taken by the scheduler to choose the next task to run.
2. Context switch time: The time required to save the current task's state and load the new task's state.
3. Cache warm-up time: The time needed to load the new task's data into the cache.
4. Dispatch latency: The time taken to actually start the chosen task on a core.

In many-core systems, scheduling latency becomes particularly critical as the number of tasks and potential scheduling decisions increases dramatically [49].

Defining Low Latency

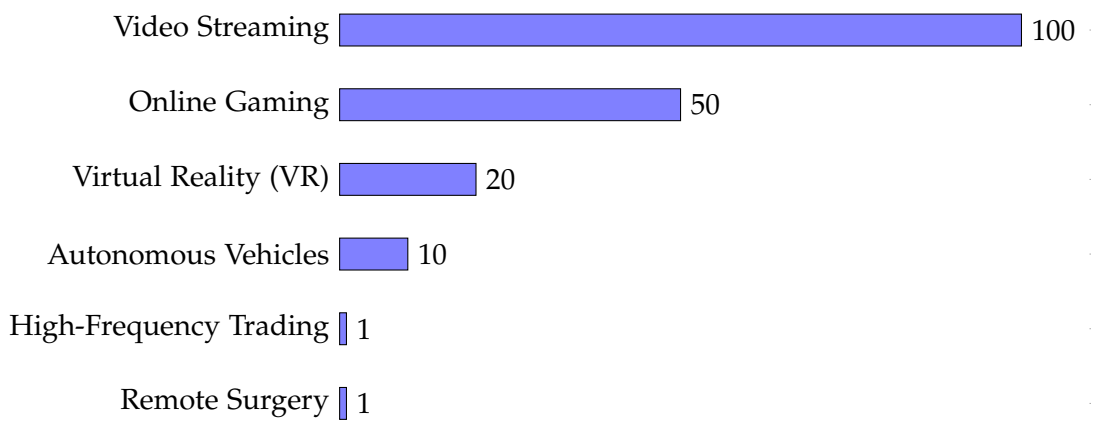


Figure 2.8.: Latency Requirements for Different Applications

"Low latency" is a relative term that depends on the specific application domain and system requirements. In the context of many-core CPU scheduling, we consider latency to be "low" when it meets the following criteria [31, 12, 13]:

1. Responsiveness: The system can react to new tasks or changes in task priority quickly enough to meet the application's timing requirements.

2. Predictability: The variation in scheduling latency (jitter) is minimal and within acceptable bounds for the given application.

3. Scalability: The scheduling latency remains low as the number of cores and tasks increases.

4. Application-specific thresholds: The latency is below specific thresholds defined by the application domain. For example:
   - For high-frequency trading systems: Latencies in the microsecond range or lower [46].
   - For real-time control systems: Latencies that are a small fraction of the control loop period, typically in the sub-millisecond range [47].
   - For interactive systems: Latencies below human perception thresholds, typically around 10-100 milliseconds[1].

5. Relative to hardware capabilities: The scheduling latency is close to the theoretical minimum imposed by hardware limitations, such as interrupt latency and cache/memory access times.

Key Techniques for Low Latency Scheduling

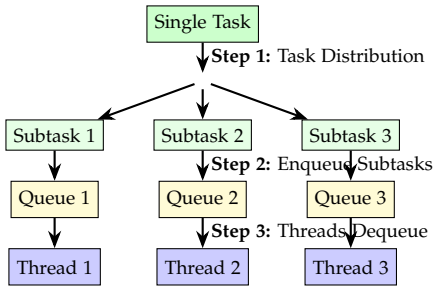
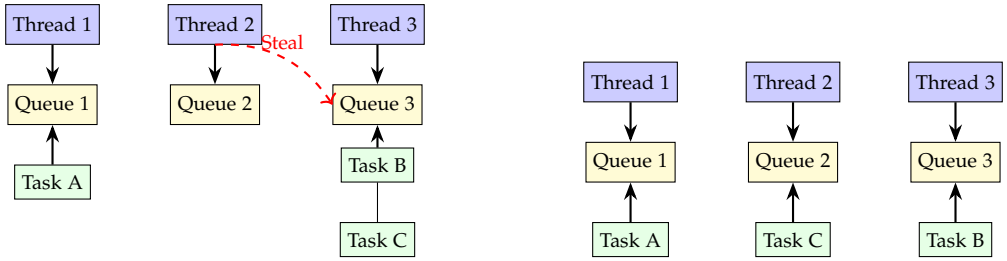


Figure 2.9.: Steps 1-3: Task Distribution, Enqueuing Subtasks, and Threads Dequeue Subtasks

Work stealing is a fundamental technique for load balancing in low-latency scheduling. It can be categorized as shared memory or task parallelism. Each processor maintains a deque of tasks, and idle processors "steal" work from others [9]. The main challenge in work stealing is the synchronization during stealing. Recent advancements and research aim to lower the synchronization overhead by locality-aware stealing [1],

[1]Results are from internet searches.

adaptive stealing [2], and different deque schemes. However, it needs to be reiterated, as proven by Attiya et al. [6], all of the synchronization cannot be removed. It is a game of trade-offs: synchronization vs. communication. Regardless of the improvements, the work-stealing approach yields slower start-up times, which are not bounded. The slowest task could be delayed as much as the whole execution. This can be the case when the thieves are repeatedly missing the task. In this case, the owning thread will execute it as the last task while many threads are idle.



(a) Idle Threads Stealing Work (b) Load Balancing Effect

Figure 2.10.: Work Stealing Overview - Stealing and Load Balancing

False Sharing Mitigation: False sharing occurs when multiple processors access different variables on the same cache line, causing unnecessary cache coherence traffic. Mitigation techniques include padding, data layout optimization, and dynamic detection and resolution [39]. The most common solution in the current implementation landscape is to pad the data to two cache lines (128B) because some architectures fetch two cache lines [24].

Lock-free and Wait-free Algorithms These algorithms eliminate blocking and reduce contention. Key concepts include Compare-and-Swap (CAS) operations, helping mechanisms, and elimination techniques [34, 30].

NUMA-aware Scheduling NUMA-aware scheduling techniques include memory page migration, task clustering (also called coalescence), and dynamic NUMA balancing through NUMA node-aware scheduling[8, 38].

Cache-conscious Scheduling Cache-conscious scheduling aims to optimize cache usage through cache partitioning, cache-aware task placement, and cache-aware load balancing [8, 38].

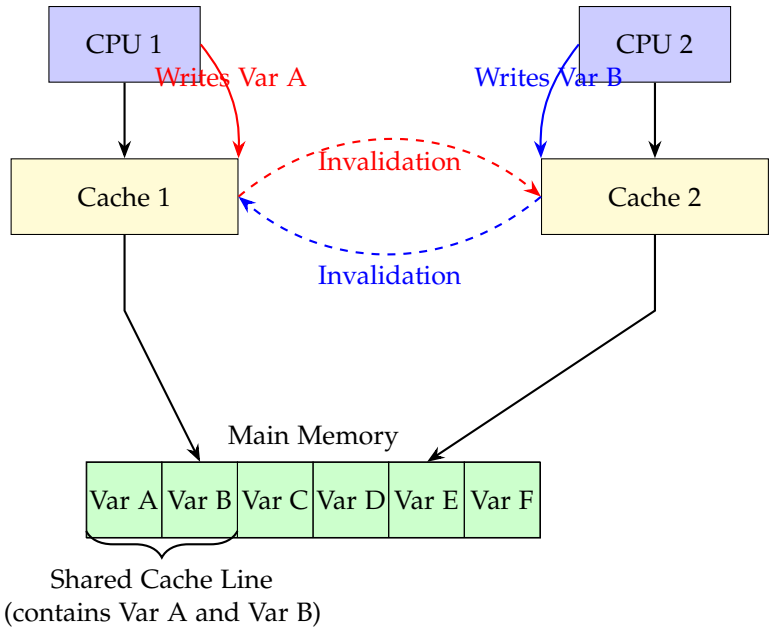


Figure 2.11.: False Sharing: Different variables on the same cache line causing unnecessary cache coherence traffic

Fine-grained Parallelism Exploiting fine-grained parallelism is crucial for low latency in many-core systems. Techniques include task decomposition, continuation-passing style, and lazy task creation [26]. The too fine-grained approach can be detrimental. For example, if the time to schedule a task is not significantly smaller than the task itself, most of the runtime will be comprised of scheduling. A common problem type is the recursive problem, such as Fibonacci. In the case of Fibonacci, the number of recursive calls also follows the Fibonacci numbers, thus growing exponentially. A cutoff point greatly improves the runtime empirically.

3. Developed Architecture

In this chapter, we present the architecture developed to address the challenges associated with standard work-stealing. We begin by illustrating a typical work distribution mechanism in work-stealing schedulers. We then explore various concurrent data structures used to optimize task scheduling, including mutex-based deques and lock-free deques like the ABP and Chase-Lev algorithms. During this section, the inefficiencies of work distribution through work stealing will be clear. Finally, we introduce the concepts of task proxies and mailboxing, which form the core of our proposed improvements.

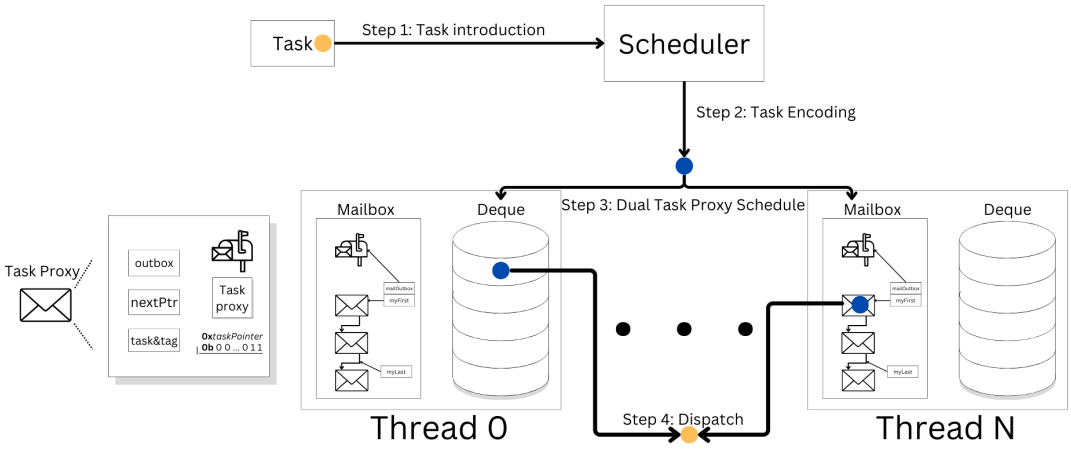


Figure 3.1.: System Design illustrating the developed architecture. From task introduction to the system to the execution of the task.

3.1. Work Distribution in Work Stealing

Work stealing is a dynamic scheduling algorithm where idle threads (*thieves*) steal tasks from busy threads (*victims*) to balance the workload across processors. As depicted in Figure 3.2, a thread initially divides its workload and makes a portion (range) available for stealing. While it continues to process its own tasks, other threads steal the available tasks non-deterministically.

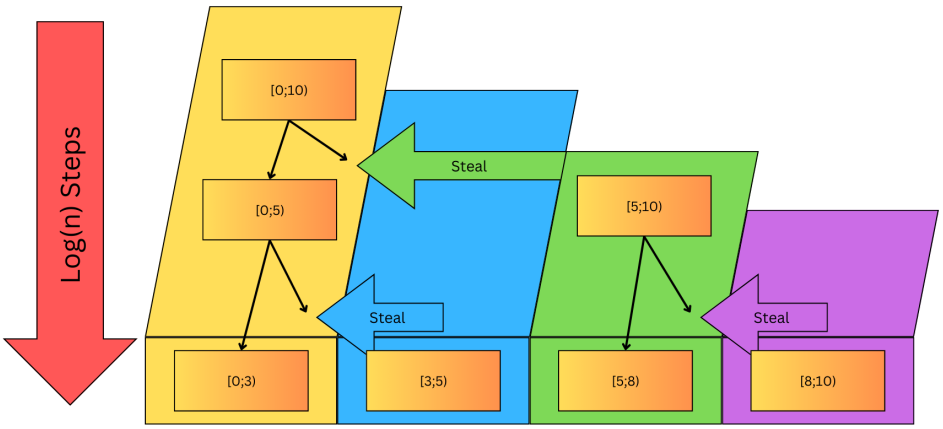


Figure 3.2.: Visualization of Work Distribution through Work Stealing[40]

In most implementations, the workload is split evenly (e.g., divided in half) to ensure that the stolen task is big enough to justify the overhead of stealing and to minimize contention with the victim's active (also called hot) data. It is important to note that exactly dividing in half will cause cache clashes on set-associative caches on powers of 2 [24]. An elegant solution is to use $mid = \frac{start+9\times(end-start)}{16}$ for the midpoint. Many schedulers rely on probabilistic models like the coupon collector's problem to estimate the expected time to visit all threads during the stealing process. However, this does not eliminate the inherent latency of work stealing: depending on stealing for work distribution. This is the first problem we are solving with our mailboxing approach.

3.1.1. Standard Work-Stealing Algorithm

To better understand these challenges, we present pseudocode illustrating the standard work-stealing scheduler.

In the WorkerThread procedure, each thread operates in an infinite loop. First, if the thread has local work, then the local work is executed; if there is no local work, the thread becomes a thief and starts to steal work. Given that there is still some work to do in the system, the steal operation will return some work; otherwise, there will be no work, and the execution will be finished.

The execution follows the divide&conquer style; if the work is bigger than the defined threshold (e.g., range size, data size, etc.), it can be divided. One part of the workload is kept(e.g., the left piece), and the rest is enqueued to its own deque, visible to thieves. This execution continues recursively until the grain size is achieved and, therefore, ready for serial execution.

Lastly, stealing tasks: defines a steal timeout, which is the time after we can conclude there is no more work; the thief tries to steal tasks within this period (amound of tries controlled by the yield factor). After unsuccessfully stealing, the thread yields its execution and waits for some work to hit the system. If the thread was able to steal a task within the timeout period, that task is returned. Else, we return a null, which concludes the execution.

Algorithm 1 Standard Work Stealing Scheduler

```
1:  procedure WorkerThread
2:      while true do
3:          if HasLocalWork then
4:              task ← GetLocalTask
5:              Execute(task)
6:          else
7:              task ← StealTask(execution_finished, timeout)
8:              if task ≠ null then
9:                  Execute(task)
10:             else
11:                 Finish
12: procedure Execute(task)
13:     if IsLarge(task) then
14:         (task1, task2) ← DivideTask(task)
15:         PutLocalTask(task2)
16:         Execute(task1)
17:     else
18:         Process(task)
19: procedure StealTask(execution_finished, timeout)
20:     start_time ← current_time
21:     repeat
22:         for i ← 0 to YIELD_FACTOR do
23:             if execution_finished then
24:                 return null
25:             victim ← SelectVictim
26:             task ← try_steal(victim)
27:             if task ≠ null then
28:                 return task
29:         yield
30:     until current_time − start_time ≥ STEAL_TIMEOUT
31:     return null
```

However, this model presents several challenges [21, 40]:

- **Non-Deterministic Stealing**: Any thread can attempt to steal from any other thread randomly, leading to many cache misses and increased latency, especially

when few tasks are available across many cores. Thieves may repeatedly target busy or already depleted deques, wasting valuable execution time

- **Inefficient for few Tasks**: When there are few tasks relative to the number of cores ($\leq log_2(\#Cores)$ because of the coupon collector's problem), the non-determinism provides no guarantee for optimal stealing, causing delays in acquiring the initial and final tasks and wasting valuable execution time.
- **High Overhead for Stealing Attempts**: Multiple unsuccessful steal attempts can occur before a thread successfully acquires a task. Depending on the data structure, failed attempts may even block other threads.

Issues with the Standard Algorithm: Work distribution through work stealing is a single producer multiple consumer (SPMC) problem, in which the consumers do not know the location of the producer. With each steal, they are guessing where a producer might be, therefore causing cache misses and ultimately not executing a part of the workload.

The number of tasks $\leq \log_2(\#Cores)$, the cutoff point for inefficiency, arises due to the Coupon Collector's Problem, which models the randomness of task distribution among multiple cores in a work-stealing scheduler. In this context, each core "steals" tasks from other cores' task queues in a non-deterministic manner. The problem states that collecting all tasks is analogous to collecting all types of coupons, where each steal attempt is like drawing a random coupon. The required number of steal attempts scale with $\mathcal{O}(n\log(n))$, n number of tasks. This number refers to the worst-case scaling when there are few tasks to do in the system. After the saturation point, we can observe a linear to logarithmic time scaling in the runtime between tasks. When there are many tasks, the probabilities are skewed in our favor. In the end, when again there are very few tasks, we can observe a sudden spike in the latency with the $\mathcal{O}(n\log(n))$ scaling. In our empirical testing, the cutoff point is at $\log_2(\#Cores)$, serving as a threshold where task stealing randomness starts to significantly affect execution time. This spike is observed twice at each execution, both in the beginning and at the end. Although possible, a mathematical derivation of this number is not necessary, as this approximate gives sufficient resolution.

The introduction of some more sophisticated scheduling than randomly selecting might come to mind. However, a more strict scheduler with more control over the system would increase the overall latency, as it would have to keep track of producers and idle processors. Task pooling, thread pooling, etc., are less efficient than work stealing

at irregular problems. Switching to less strict execution with async or futures also does not solve this issue, as they, too, introduce latency with their own thread management schemes. The most optimal way forward is to solve the scheduling latency of work stealing without significant overhead while maintaining a better cache locality. These inefficiencies highlight the need for improved work distribution mechanisms, which we address through advanced concurrent data structures and optimized scheduling strategies.

3.2. Concurrent Data Structures

Another problem is the lock contention during the steal procedure itself. Stealing requires synchronization. If multiple thieves are targeting the same victim, all but one of the thieves need to wait. Often, this is not the only problem; the work introduced by the victim to its own deque, contends with the thieves, incurring unnecessary waiting. To tackle this problem, many concurrent deque implementations were developed, and their bounds were also proven. In our architecture, reducing the latency caused by stealing is also targeted. We have implemented many types of deques, which have been proposed by different papers. For our purposes, we need three methods: pushBottom, popBottom, and popTop. Operations acting on the bottom are called by the owning thread, while popTop (also called steal elsewhere) can be called by any thread.

3.2.1. Concurrent Deque with Mutex

The concurrent deque with mutex may be the simplest of all of the deques used in work stealing. This type of deque does not rely on the memory model of the hardware but the efficiency of mutexes. The program needs to manage a pair of deque and mutex. During enqueue and dequeue, the mutex is locked, preventing concurrent access. While work stealing with concurrent deques is proven to be efficient, this approach suffers from two limitations [2]:

- Local deque operations also require expensive synchronization (memory fences, barriers)
- They are hard to extend to support optimizations for irregular problems

Regardless, they are still being used in modern frameworks, such as oneTBB by Intel. By using normal deques, they can specify whether to execute with LIFO or FIFO. While not strict-fork-join, this approach helps to load balance in irregular problems by enabling dependency-reserving dynamic execution (flow-graph) [21]. In our architecture, this approach was less than optimal because of the latencies introduced by the locks. Therefore, we opted not to use concurrent deques with mutexes.

3.3. Lock-free Deques

The lock-free design emphasizes the removal of explicit mutexes through atomic operations. The most prominent of these operations are the CAS and memory fence operations. These lock-free structures aim to reduce contention and improve performance, especially in high-concurrency scenarios. They are essential for low-latency performance.

3.3.1. The ABP Algorithm

The ABP algorithm, named after its creators Arora, Blumofe, and Plaxton [5], is a foundational concurrent deque implementation designed to optimize work stealing in parallel computing environments, as it is the first lock-free concurrent deque algorithm that has proven bounds. It serves as a critical component in modern work-stealing schedulers, enabling minimal synchronization overhead and reducing contention among threads.

Design and Operation

The ABP deque employs an asymmetric access pattern where the owning thread (the *worker*) operates on the *bottom* of the deque, while other threads attempting to steal work (the *thieves*) interact with the *top*. This separation allows the worker to perform push and pop operations with minimal synchronization, leveraging the fact that local access patterns dominate typical workloads.

The deque maintains 2 fields:

- `age`: is a struct encapsulating top and tag. Top points to the top of the queue, where thieves can steal. Tag is not a pointer; it is used for race conditions during pop operations.
- `bottom`: Points to the position one below the last task.

Tasks are stored in a static array, allowing efficient and fast use of memory. The field bottom and the struct age must be able to be modified atomically without locks. This means in modern computer architectures, the maximum size for age is 64 bits. Although `Atomics Library` allows for larger structs, internally, they are using locks [24].

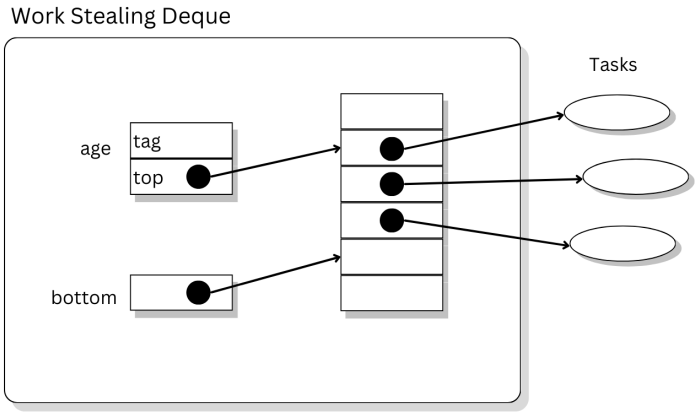


Figure 3.3.: Visualization of the Deque [5]

Operations

The ABP algorithm defines three fundamental operations: pushBottom, popBottom, and popTop. We present the pseudocode for each operation and explain their implementation details.

pushBottom Operation: It allows the worker to add a task to the bottom of the deque.

Algorithm 2 ABP Algorithm pushBottom Operation

```
1: procedure pushBottom(Task* task)
2:     localBot ← bot
3:     deq[localBot] ← task
4:     localBot ← localBot + 1
5:     bot ← localBot
```

In this operation, the worker stores the task at the corresponding position in the array and increments the `bottom` index. Since the worker is the only thread modifying the `bottom` index during push operations, no synchronization is required.

Algorithm 3 ABP Algorithm popBottom Operation

```
1:  function popBottom
2:      localBot ← bot
3:      if localBot = 0 then return NULL
4:      localBot ← localBot - 1
5:      bot ← localBot
6:      task ← deq[localBot]
7:      oldAge ← age
8:      if localBot > oldAge.top then return task
9:      bot ← 0
10:     newAge.top ← 0
11:     newAge.tag ← oldAge.tag + 1
12:     if localBot = oldAge.top then
13:         CAS(age, oldAge, newAge)
14:         if oldAge = newAge then return task
15:     age ← newAge return NULL
```

popBottom Operation: popBottom operation enables the worker to remove a task from the bottom of the deque. It is used by the owner thread to retrieve work from its own deque. It's the most complex of the three, as this operation is in conflict with the stealing operation. We begin by checking whether the queue is empty; in that case, we return null. Then, the next lines are the usual pop operation, in which we retrieve the task. If the index of the retrieved task does not overlap with the top index (i.e., not the last task), we can safely return the task without the risk of race conditions. If not, then we need to contend with a potential thief. There are two mechanisms to use: memory fences or CAS operation. In this case, for correctness, we need to use a threadfence and **strong CAS**. Though these operations are strictly lock-free, they may not be wait-free [5].

popTop Operation: This operation is used by thief threads to steal work from the top of another thread's deque. Its process is similar to popBottom, without the need to manipulate the bottom index. If there is a task in the queue (checked in l.4), the task is retrieved and stored locally. Then, the stealing thread atomically updates the top pointer. If it succeeds, then we can return the local task, else we return null. The role of tag might not be obvious at first sight, however, in a scenario where the thief thread was preempted after executing line 5, the tag becomes critical. This means the local task has been retrieved, but the queue has not been modified to reflect that. While

Algorithm 4 ABP Algorithm popTop Operation

```
1:  function popTop
2:      oldAge ← age
3:      localBot ← bot
4:      if localBot ≤ oldAge.top then return NULL
5:      task ← deq[oldAge.top]
6:      newAge ← oldAge
7:      newAge.top ← newAge.top + 1
8:      CAS(age, oldAge, newAge)
9:      if oldAge = newAge then return task
10:     elsereturn nullptr
```

the thief waits, the owner thread popped a task and pushed another task. When the execution of the thief continues, the CAS will succeed, thus logically removing a task from the queue. In this situation, the thief has an invalid task, and a task has been lost. To mitigate this, the tag will be updated during popBottom (l.11); this way, the CAS will fail for the thief. Necessitating a new steal procedure. In the original paper, the authors also allow the return of a special value called ABORT when the thief loses a race with another thread [5].

Place of ABP Algorithm in our architecture is undeniably important. In a head-to-head comparison with the mutex variant, the ABP is at worst case 10% and, at best case, 8 times faster latency-wise and scales better due to no lock contention over the shared queue. These findings are also in line with recent benchmarks comparing mutex-based and lock-free queue algorithms for memory-intensive problems [7]. Unfortunately, the speed also has a cost: static size and synchronization, even for local operations.

3.3.2. Chase Lev Algorithm

The Chase-Lev algorithm, introduced by David Chase and Yossi Lev in 2005 [16], is an improvement over the ABP algorithm, offering a dynamically sized work-stealing deque. Dynamic size is not necessary for many families of problems, yet for problems with high branching or short bursts of tasks, the static queue size might overflow. Because of the hybrid nature of our scheduler, dynamic queue size is a requirement for those problems. Modern frameworks also employ either a mutexed deque (e.g., OneTBB) or a dynamic deque based on Chase Lev (e.g., CILK) to combat static size.
In the following section, we will present the changes to the ABP algorithm, supplying pseudocode and visualizations.

Design

Similar to ABP, Chase-Lev also employs asymmetric access to both ends of the queue. There are notable differences: The Deque manages only the top and bottom pointers without an additional tag field. The internal queue itself is now a circular **dynamic** array.

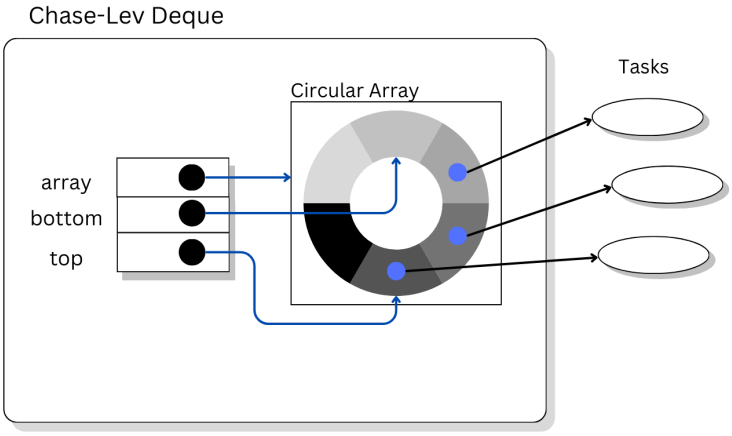


Figure 3.4.: Visualization of the Deque [16]

Operations

For the sake of brevity, we will not go in-depth at every operation, as they are very similar to the ABP algorithm [1]. In this section, we will highlight the differences and their impact on our architecture.

PushBottom Operation remains almost exactly the same with one difference. At maximum capacity, this operation will double the circular array size and copy the tasks. Afterward, the new task is inserted at the bottom position and incremented at the bottom. For memory management reasons, one cell of the array always remains unused [16].

[1]The reader can find the more in-depth algorithm description in the original paper [16]

PopBottom Operation also remains mostly similar. Without the tag field, the synchronization is solely dependent on the top field. This works because the top field is only incremented. Because of the dynamic nature of the deque, we need to keep track of its size more carefully. If we only let it grow, there would be a lot of memory wasted. Therefore, we can also shrink the array with this method. The threshold for shrinking is `arraySize`$/K$, K is a constant $3 \leq K$. PopTop Operation is exactly the same [16].

Chase Lev Algorithm also proposes a more complex and efficient memory use. Instead of growing and shrinking every time, we logically keep the indexes for growth and shrinkage. This way, we save time on allocations. Allocating a bigger memory at the start of the execution and working on the allocated memory hides the latency of system calls. In our case, even though growing and shrinking are rare events, reduced latency is an important factor.

One of the biggest downfalls of Chase-Lev is the mandatory reads to the `top field`. For every method, we need to read the value of the top. For bottom-field operations, it is to calculate the size, therefore deciding to either grow or shrink and for top-field operations, it is for the correctness of the steal. This means we cannot make use of the memory reorderings, thus necessitating expensive thread fences. However, our and external benchmarks suggest that ABP and Chase-Lev have roughly the same runtime characteristics for small to medium-sized computers (8-16 threads). As the number of threads increases, the throughput of ABP surpasses the Chase-Lev. Conversely, in multiprogrammed environments, Chase-Lev scales better than the ABP [16]. Still, with the Chase-Lev deque, local operations to the bottom of the queue require more synchronization than the ABP due to thread fences.

3.3.3. Split Deques

Building upon the limitations identified in the previous algorithms, we explore the concept of **split deques** as an effective solution to reduce synchronization overhead during local operations. This approach is based on the works of Acar, Charguéraud, and Rainey [2] and Rito Silva et al. [45], which propose minimizing synchronization by separating the deque into private and public regions.

Design and Motivation

In the ABP and Chase-Lev algorithms, synchronization is required even for local operations on the bottom of the deque to ensure correctness when interacting with thieves. This synchronization introduces unnecessary overhead, especially when the

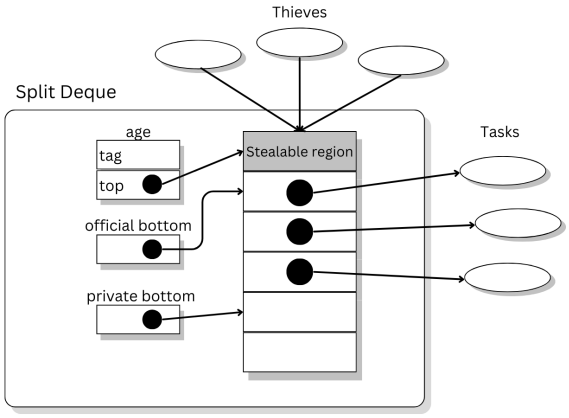


Figure 3.5.: Illustration of Private and Public Deques

majority of operations are performed by the owner thread.

The idea behind private deques is to partition the deque into two distinct regions:

- **Private Region**: Accessible only by the owner thread without any synchronization.
- **Public Region**: Accessible by both the owner and thieves, requiring synchronization to maintain consistency.

By ensuring that the owner thread performs most of its operations in the private region, we can eliminate synchronization overhead for these operations. Stealing threads (thieves) can only access the public region, and a mechanism is provided for the owner to transfer tasks from the private region to the public region when **a request** is received.

Operations

In addition to the standard operations, we have operations that only operate in the private region. As well as an operation to transform a task from private to public region for load balancing. In total, the split deque describes 5 operations. `Push`, `Pop`, `Transport`, `PopBottom`, and `PopTop`.

Private Operations The worker thread has uncontested access to the private part; therefore, these operations are completely synchronization-free. The owner thread pushes and pops tasks onto its private deque without any synchronization. This way

the number of fences and CAS operations are significantly reduced. A 2023 paper comparing work stealing with split deques showed that the number of memory fences is reduced by a factor of 1000, while the number of CAS operations is halved [18]. Due to the simplicity of these operations, we will spare the implementation details here. They can be read in the appendix.

Public Operations If the private deque is empty, the owner attempts to pop from the public deque's bottom. Popping from the public deque requires synchronization. In essence, the public region is the ABP algorithm with more than one bottom index. The implementation is mostly the same, but as a design choice, we chose to make the `popTop` operation fence free and pushed the overhead to the `popBottom` by using two fences. This is beneficial to the runtime, as there are more steals happening than self-public region access.

Advantages and Implications

The use of private deques offers several advantages. As described above, without synchronization, local operations are faster. By limiting work migration to other cores and potentially other NUMA nodes, we can improve the cache locality. However, the work distribution with split deques is significantly slower. Two-step stealing limits the window for work distribution, as only one work per worker loop iteration is set public. Intuitively, this approach does not fit for low latency scheduling. This is correct, but our scheduler does not only rely on work stealing for work distribution. Our scheduler uses a hybrid mode of work distribution, therefore limiting the need for stealing at the start and end of the execution. After this phase, split deques offer better performance in the saturated phase of the program execution[2].

3.4. Mailboxing

Mailboxing represents the crux of our implementation. This is where we push the limits of work stealing by introducing dual scheduling. In this scheme, a task is offered in the system through two channels, allowing faster access to the task. As discussed in the work distribution chapter, non-determinism does not necessarily deliver the best performance during the start and end of the execution. We are giving the system a determinate and a non-determinate way of getting the task. This way, we guarantee low latency goals for the start of the execution with higher probability. The main idea

[2]Saturated in this context means that all of the threads are executing. In contrast, program start and end do not qualify for saturated state.

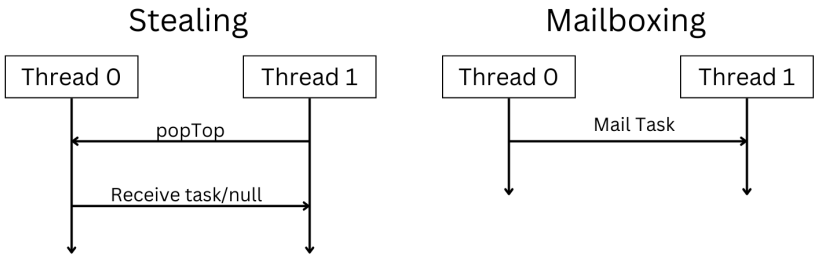


Figure 3.6.: Illustration of Steal and Mailbox

behind dual scheduling lies in the fact that sending a signal is faster than receiving it. A thread can send a pointer faster than the time it takes to steal it from its queue. This is self-evident, as the stealing procedure has both sending (popup function call) and receiving (return value) components. However, the overhead of selecting the `idle` thread to send work to is more than work stealing [2]. This is why we do not keep track of idle threads but send them in a manner similar to work stealing's victim selection. This way, with low overhead, we can exploit the best of both work stealing and work sharing.

We will explain the changes to the scheduling algorithm, the building blocks of mailboxing, and different strategies for mailboxing. Our implementation is based on the OneTBB implementation.

The changes to the scheduler algorithm are minimal. In addition to the mailbox structure to the `threadData`, there are no structural differences. In the main loop, we encapsulate the task into taskProxies and mail it to another thread's mailbox. We are keeping the option to push local work to our own deque, as this allows for stealing later in the execution. The final change is during the retrieval of the tasks. For local task access, we first check the mailbox for potential work, then the deque, and eventually stealing. At the start of the execution, all of the threads are devoid of work to do. As a result, all of them become thieves. This is detrimental to mailboxing. Premature stealing causes unnecessary work migration and ultimately defeats the purpose of work sharing. It also breaks the cache locality and can artificially create load imbalance. To solve this, we utilize a flag `canSteal`. Threads are in the waiting state until they can steal. After a number of tasks have been mailed or after a certain time passes, the flag is set. The numbers are variable, and as long as they are not shorter than the average scheduling latency + inter-task scheduling latency, premature stealing will be avoided. Delaying the ability to steal might ultimately hurt the load balancing if the mailbox is not uniform on all threads, creating an imbalance in the system without the ability to solve it.

Algorithm 5 Work-Stealing Scheduler with Task Encapsulation and Mailboxing

```
1: procedure WorkerThread
2:     while true do
3:         if HasLocalWork then
4:             task ← GetLocalWork
5:             Execute(task)
6:         else
7:             task ← StealTask(execution_finished, time-
   out)
8:             if task ≠ null then
9:                 Execute(task)
10:            else
11:                Finish
12: procedure Execute(task)
13:    if IsLarge(task) then
14:        (task_1, task_2) ← DivideTask(task)
15:        proxy_2 ← EncapsulateTask(task_2)
16:        MailTo(proxy_2, someThreadID)
17:        PutLocalTask(proxy_2)
18:        Execute(task_1)
19:    else
20:        Process(task)
21: procedure GetLocalWork
22:    if MailboxNotEmpty then
23:        taskProxy ← Mailbox.pop()
24:        task ← extractTask(taskProxy)
25:    else
26:        taskProxy← Deque.pop()
27:        task ← extractTask(taskProxy)
28:    return task
29: procedure StealTask(execution_finished, timeout)
30:    while !canSteal do yield
31:    start_time ← current_time
32:    repeat
33:        for i ← 0 to YIELD_FACTOR do
34:            if execution_finished then
35:                return null
36:            victim ← SelectVictim
37:            task ← ExtractTask(try_steal(victim))
38:            if task ≠ null then
39:                return task
40:        yield
41:    until current_time - start_time ≥ STEAL_TIMEOUT
42:    return null
```

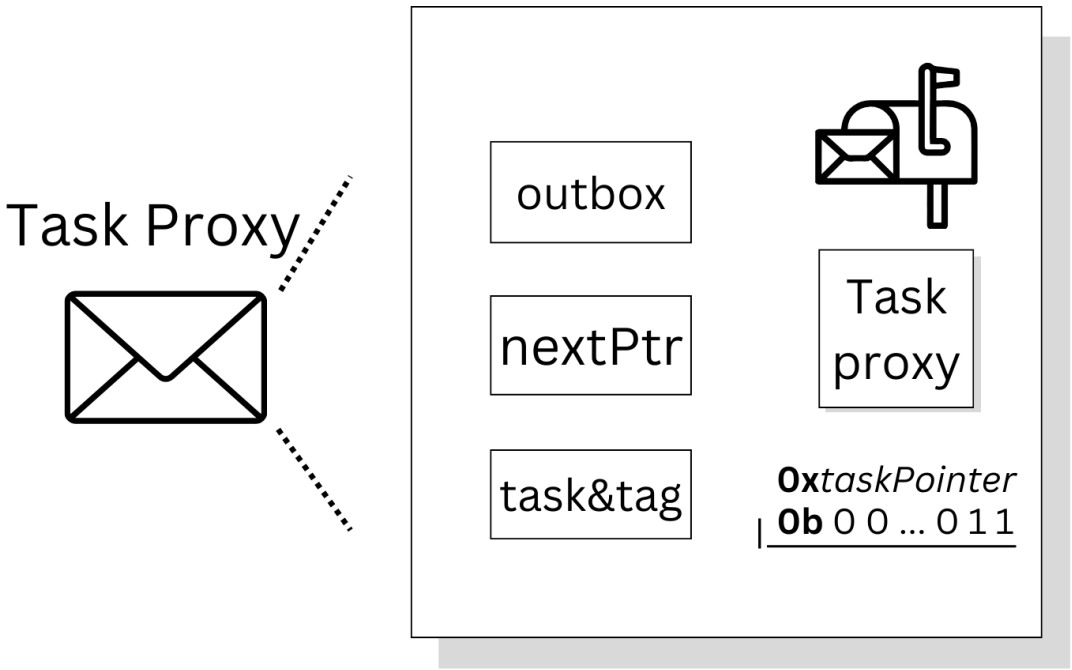


Figure 3.7.: Structure of Task Proxy

3.4.1. Task Proxy

For dual scheduling, we need to encapsulate the task in a proxy. This is because of the nature of execution. If there are two instances of the task, after one instance is successfully executed, the other instance is still in the system waiting to be scheduled. There are multiple problems in this scenario. First, double execution of the same task will result in a faulty result[3]. Directly invalidating the task by `deleting` will result in segmentation faults.

A good solution is to divide the execution and deletion of the task. The faster thread will execute the task, and the slower thread will delete it. In the given pseudocode [5], all of the tasks in the system are encapsulated in task proxies. Operations to encapsulate and extract are inexpensive and comparable to interacting with the deque itself. In this sense, we did not increase the complexity. The cost of task proxies will be later investigated. Task proxy, as seen in figure 3.7, has 3 important fields:

- **Outbox**: This is the mailbox to the proxy that will be mailed.

[3]If the operation is not idempotent.

- **nextPtr**: For queue-like order in the mails in the mailbox.
- **task&tag**: Masked task pointer.

We define three masks: $0x1$ for deque, $0x2$ for mailbox, and $0x3$ as location mask. In the encapsulation, we mask the address of the task with the location mask. During the extraction, supplied with one of the masks (l. 5), the faster thread gets the underlying task. Another thread is left with the empty proxy, which it can promptly delete.

Algorithm 6 EncapsulateTask and ExtractTask Pseudocode

```
1:  procedure EncapsulateTask(task)
2:      proxy ← allocator.new_object<task_proxy>()
3:      proxy.task_and_tag ← (address of task) | location_mask
4:      proxy.outbox ← scheduler.mail_outboxes[target_id]
5:  procedure ExtractTask(fromBit)
6:      tat ← task_and_tag.load(memory_order_acquire)
7:      if tat ≠ fromBit then    ▷ Compute the cleaner bit by masking off the from_bit
8:          cleaner_bit ← location_mask & ~from_bit
9:          if task_and_tag.compare_exchange_strong(tat, cleaner_bit) then
10:             return task_ptr(tat)                          ▷ Unmasked task pointer
11:     return null
```

3.4.2. Mailbox

Mailbox is the counterpart of the deque. It is the container responsible for holding the task proxies. Proxies are stored by chaining and then using the `nextPtr` field. We introduce two types of mailboxes: `mailInbox` and `mailOutbox`. MailOutboxes are locally stored in the scheduler object, while the mailInboxes are in the `threadData`. MailInboxes are wrappers for the mailOutbox objects, granting access to internal operations. Separation of mailboxes is a cache-friendly way to give access to the same object with different localities. Mailbox has 3 important methods: `pop`, `push`, and `empty` acting on atomic proxy pointer myFirst and **pointer to pointer** to task proxy myLast.

Algorithm 7 Push Pseudocode

```
1:  procedure push(task_proxy t)
2:      t− > next_in_mailbox ← null
3:      link ← my_last.exchange(&t.next_in_mailbox)
4:      link.store(t, memory_order_release)
```

Push Operation: This operation is fairly easy. We set the next pointer of the new task proxy to null and exchange the `myLast` with the address of `nextPtr` of the new task proxy. Finally, atomically storing the changes. There is no need to look for race conditions, as we don't want to busy the thread sharing the task. The responsibility is shifted to the thread owning this mailbox. Making the thread wait that is sharing the work would incur unnecessary waiting times for future work distribution, ultimately hurting the total latency. Thread without work is already in the waiting state; waiting more will not significantly introduce latency.

Algorithm 8 Pop Pseudocode

```
1:  procedure INTERNAL_POP
2:      curr ← myFirst.load(memory_order_acquire)
3:      if curr = null then
4:          return null                                          ▷ Mailbox is empty
5:      prev_ptr ← &myFirst
6:      second ← curr.next_in_mailbox.load(memory_order_acquire)
7:      if second ≠ null then          ▷ There is more than one task, we can pop it easily
8:          prev_ptr.store(second, memory_order_relaxed)
9:      else                           ▷ This was the last task, we need to reset the mailbox
10:         prev_ptr.store(null, memory_order_relaxed)
11:         expected ← &curr.next_in_mailbox
12:         if myLast.compare_exchange_strong(expected, prev_ptr) then     ▷ Now it's empty
13:         else
14:             while second = null do
15:                 Yield thread and wait for other thread to complete
16:                 second ← curr.next_in_mailbox.load(memory_order_acquire)
17:             prev_ptr.store(second, memory_order_relaxed)
18:     return curr
```

Pop Operation: This operation is simple, yet one part needs to be highlighted. Strictly speaking, there is only one scenario where race conditions can occur in the mailbox. It is when a foreign thread pushes a task while the owner thread pops the last mail in the mailbox concurrently. Instead of memory fences, we are choosing to wait. Exponential backoff strategies can also be beneficial, yet this race condition is exceedingly rare, and simply waiting is sufficiently fast. The rest of the operation is a `linked list pop operation`.

The mailbox data structure has $\mathcal{O}(1)$ complexity, similar to the deque. This is also self-evident, as the operations are changing only the head or the tail of the queue. Time complexity substantiates our claims of low overhead introduced by mailboxing.

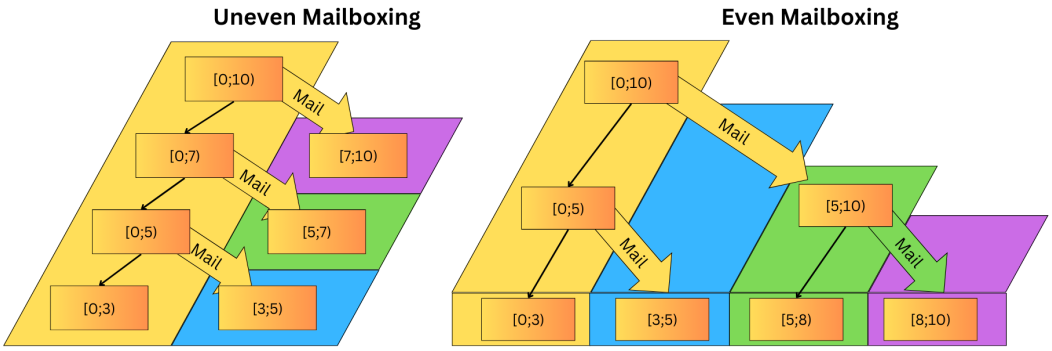


Figure 3.8.: Visualization of Uneven and Even Mailboxing [40]

3.4.3. Mailboxing Strategies

With building blocks explained in depth, we can discuss the ways in which mailboxing can be utilized. There are two ways to use mailboxing: one using the affinity method similar to work stealing, where the propagation of work follows a balanced binary tree. We call this method `even mailboxing`. This follows a similar split-in-half partitioning, such as work stealing. This way, the work affinity (data locality) is mostly preserved, unlike work distribution through work stealing. The work distribution is deterministic, as the number of tasks and cores are known beforehand. If the number of jobs is not known (e.g., recursive algorithm, parallel invocation of lambdas), even then, the tasks are distributed systematically according to the even mailboxing scheme. In essence, we are rigidifying the work distribution, eliminating the latency induced by non-determinism. `Even mailboxing` improves the startup times compared to classic work stealing, yet it is not optimal.
Logically, the other way is called `uneven mailboxing`. In this variant, the work propagation looks like a lopsided binary tree. Earlier threads produce the vast majority of the work, reducing the overall communication in the system early on. The Conservative version defines a single producer, and all other threads are consumers. Time to start is significantly reduced, as mailed tasks are **execution ready**. Tasks are divided in

proportion to the number of cores. This also means the task will not move around, causing cache misses, as is the case with stealing. Loaded memory will be executed, improving also the cache performance. Combined, this scheme yields even better performance for task start time. It is only logical; making executable work available earlier will result in earlier execution.

Uneven work stealing will not work because the benefits of `deterministic mailboxing` are not part of the classic work stealing. Dividing work in proportion to the number of threads and making it available in one's own deque is not a good idea. The producer thread will fill its deque, becoming the only thread with work. Centralized work will necessitate more stealing, which needs to use expensive popTop operations. Together, this causes more start latency.

We can compare uneven mailboxing to writing multithreading by hand. Explicitly defined work and memory. Then, starting threads in a for loop, rapid-fire start. However, this does not make the execution significantly faster than using a library or automatic multithreading[4].

3.4.4. Morsel

A bonus section dedicated to cache performance. This section is based on Leis et al. 2014 [37]. While their work focused on query execution in database systems, we adapted this approach to enhance cache locality and mitigate false sharing in our task-based parallel execution model. Tasks in our architecture are C++ objects that have overloaded the () operator, acting like C++ lambdas without parameters and no return type. This means any action can be a task. System calls, print statements, or `doesSomethingUsefull()`. While this approach allows us to do any type of work in the system, we need to be careful when using memory this way. Memory residing at the location of the job called `task` will be loaded into the active cache as soon as its pointer is accessed. Now, the execution per `task()` needs to eventually load more memory, causing memory-related latency. This is usually not a problem. As discussed in the background, CPU hardware has developed multiple coping mechanisms for memory-related waiting. However, we can also help in this case by using OneTBB's blocked ranges. The advantage of blocked ranges, as opposed to normal data in the form of pointers, is memory guarantees. The wrapped range will be aligned to the cache line; it will mitigate false sharing and implement iterators for the value type. Overall, improving the cache performance. Input data is converted into blocked ranges, delivering faster performance and fewer cache misses when using the partitioning

[4]For simplicity's sake, correct OMP multithreading.

methods. However, this operation is not automatic in our architecture. Automatic memory optimization and migration to thread local memory would improve the cache performance. This is still an area in development and part of the future research. Our preliminary results and findings by the original author hint at better scalability in memory-intensive workloads.

4. Evaluation

In this chapter, we will discuss the performance metrics of our architecture. Starting with the test setup, we will compare the classical work stealing to our hybrid approach. Deque implementations will be evaluated, and their cache, memory fence, and CAS performance will be analyzed. Lastly, we will present the overhead related to Mailboxing.

4.1. Test Setup

Our test setup consists of two systems.

- **AMD16**: Ryzen 9 5900hs (8C, 16T) @ 4.6GHz, Pop!_OS 22.04 LTS x86_64, Linux kernel 6.8.0, Clang 17, 32GiB DDR4 @ 3200 MT/s
- **Cloud128**: 128 virtual cores @ 2.8GHz, Ubuntu 22 LTS, Linux kernel 6.8.0-1012-aws, Clang 18, 128GiB

Our main goal is to show low latency on many-core CPUs; this corresponds to Cloud128. As a sanity check, we also test on consumer hardware. Cloud computing assigns cores on demand. This is less than ideal for scheduling latency measurements, as these numbers are in the microsecond range. However, we think the numbers from Cloud128 illustrate the underlying performance gains. A concern on AMD16 platform is the cache layout on the CPU design, more precisely, the NUMA nodes. In the chiplet design of 5900hs, there is a numa node per 8 threads. Memory access in the L3 shared cache from one node to another is significantly slower. Circumventing this effect on this platform is particularly hard, as a manual NUMA-aware victim selection function is needed.

4.2. Work Stealing vs Mailboxing

4.2.1. Benchmarks

This section serves to showcase the competitive performance of our hybrid scheduler. We have tested `cilksort`, `fibonacci`, `knappsack`, `matrix multiplication`, `Strassen`

`algorithm`, `Monte Carlo Pi Approximation`, and `N-Queens`. These tests form a good basis for both recursive algorithms requiring parallel invocation and parallel for loop algorithms.

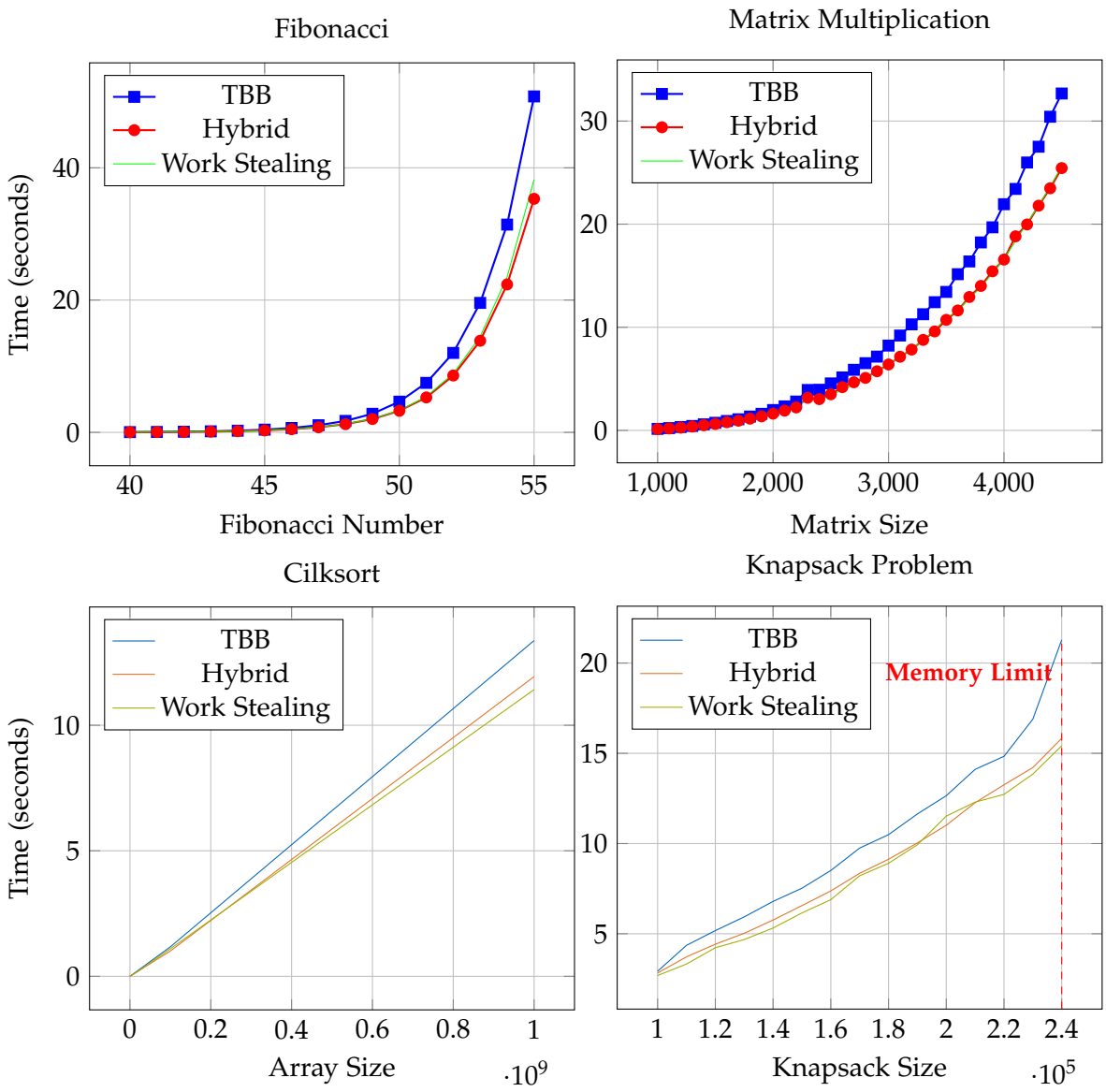


Figure 4.1.: Performance comparison between Hybrid, Classical Work Stealing, and OneTBB schedulers

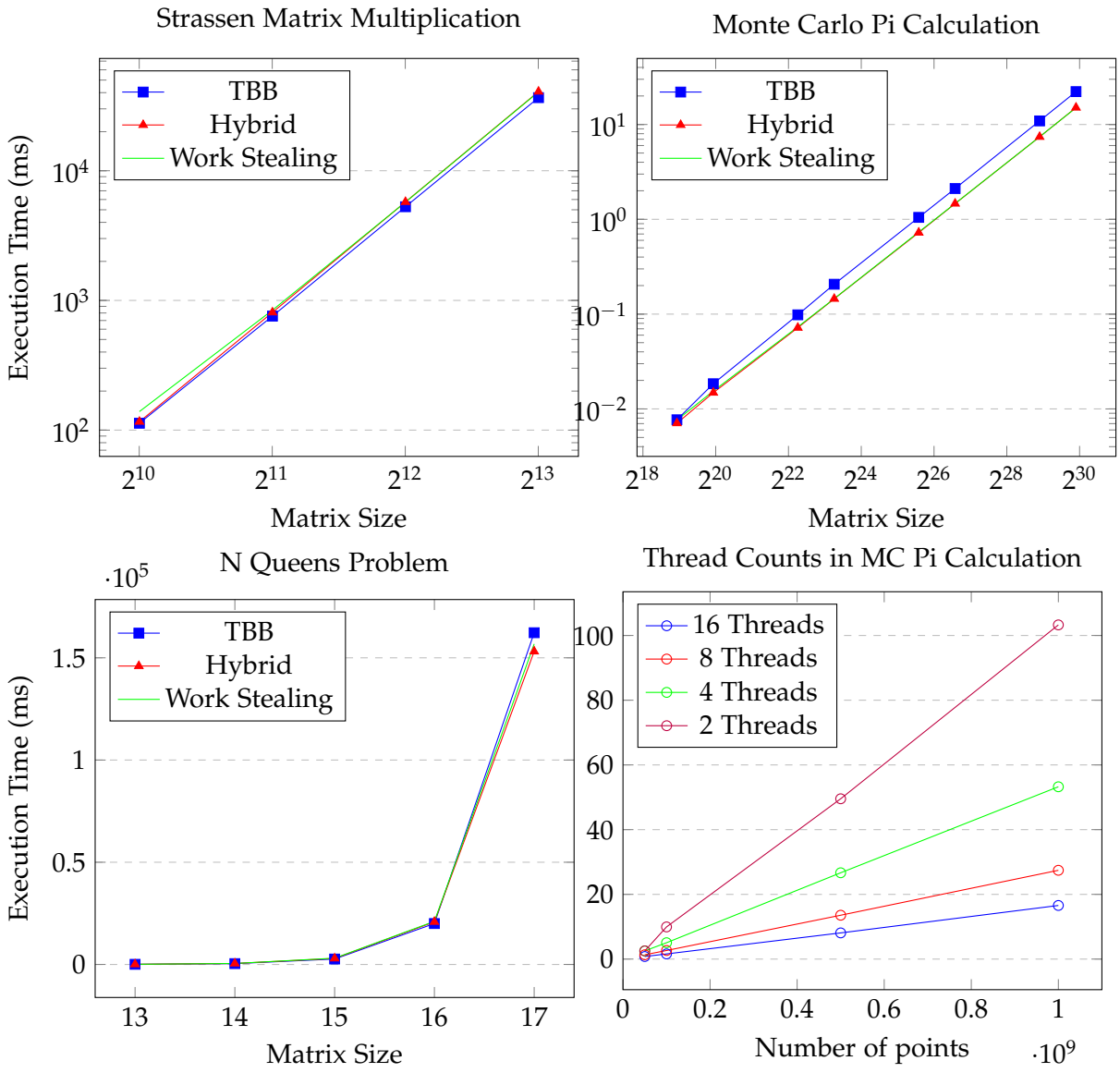
Figure 4.1 and 4.2 showcase a consistent performance compared to the TBB and classical work stealing. Per the problem, we averaged 10 runs after running 1 iteration to heat the cache. We used optimization level 2 of the Clang compiler. GCC delivered similar results. Intel's own compiler `icpx` gave overall better performance on the AMD16 system. Unfortunately, our cloud system was unable to use icpx due to problems related to the provider.
In memory-intensive problems like `matrix multiplication` or `cilksort`, both of our scheduler implementations beat the OneTBB scheduler. This is thanks to our morsel-based work partitioner. However, in `Strassen Algorithm`, this difference is not observed. We think it is less pronounced, likely due to the divide and conquer algorithm executing on smaller subsets of the matrix. Therefore, the need for memory, as well as any latency caused by bad memory handling, is smaller than normal matrix multiplication. In problems necessitating greedy solutions, `knapsack` and `N-Queens`, our scheduler is on par with the OneTBB implementation, even leading by small percentiles. These small differences can be explained by run-to-run variation.

In Fibonacci calculation, our scheduler is faster than OneTBB beyond, which can be explained by run-to-run variation. This test is especially important because of its exponential growth. Fibonacci can be used to demonstrate fine-grained task parallelism. The number of tasks in the system likewise increases exponentially. During this testing, we experienced the limits of the ABP Algorithm, which often overflowed during bigger Fibonacci number calculations. This was not a load balancing issue, as the average utilization of cores was 99%. The efficiency of work distribution by mailboxing caused overall less stealing in the system, not just in the beginning. Not stolen tasks clutter the deque, waiting to be deleted. This does not cause performance issues but shows inefficiencies in memory management. This issue is solved with the Chase-Lev algorithm. These results show that our scheduler performs well in irregular problems and in problems requiring fine-grain parallelism.

Lastly, we examined the scaling of our scheduler. We deemed the Monte Carlo Pi Calculation appropriate for showing this because of its embarrassingly parallelizable nature. Doubling the core number halves the runtime. Only at the last doubling (from 8 to 16) do we see around 60-70% improvement, likely due to running out of real cores (8 cores, 16 threads). Cloud128 system yielded similar results, providing evidence for scalability.

We conclude our improvements do not hurt our performance. The performance difference between the hybrid implementation and the classical work stealing is cumulatively 1%, essentially delivering the same performance.

Figure 4.2.: Performance comparison between Hybrid, Classical Work Stealing and OneTBB schedulers



4.2.2. Latency

This section provides evidence for our core promise of low-latency scheduling. Latency, as defined in the background section, was calculated by measuring the time between

the task creation (also called `task introduction`) and its execution. There are technical difficulties in measuring the time accurately: system calls. Writing code that leverages the highest resolution clock available will necessitate a system call to access the system_clock (or the steady_clock in Windows). System calls are inherently unwanted during performance and latency measurements. There are two solutions to this problem. We can use the `rdtsc register` to obtain a sufficiently high-resolution measurement. Using this approach does indeed give low overhead, but it can be unreliable, delivering inconsistent data, because of CPU frequency scaling, hibernation etc. Intel claims to deliver "consistent and reliable" rdtsc with newer CPUs. However, this needs more care than they claim. Two problems again: out-of-order execution and clock speed. As discussed in the background chapter, OOE will deliver a clock reading that **might** be executed out of order. That is why we need to execute a serializing call (the Intel guide suggests 'CPUID') before the clock reading. The second problem is the suitability of the TSC counter itself, having unstable TSC counters. This is especially a problem in multi-processor (or multicore) systems, where the clocks are not synchronized to the same value[1]. With these implications in mind, we decided to move with the second option.

The second option for accurate measurement is making the task itself the system call. This option delivers a general-purpose way to measure latency across platforms and libraries. We will measure the time for task starting times and the time between tasks. This is a broad generalization of the latency definition, where the task introduction is the execution start. Every subsequent time measurement assumes all tasks are created at the start. Alltogether for $task_i$ the latency is calculated as: $t_{start} - t_{execution\ of\ i}$. Formulated like this, we can also measure the latency of OneTBB.
Last considerations before data evaluation. There are some calculations that need to be done to acquire accurate time measurements using the hybrid scheduler. Particularly, the creation of the scheduler itself is expensive because of the many additional constructions that scale with the number of cores. Mailboxes, scalable allocators, etc., are created on the `heap`. The time to construct is negligible in usual calculations, but measuring scheduling latency in the sub-millisecond range needs to be distinguished from the construction overhead.

The nature of this test is highly susceptible to changes in the underlying system because it is very short. Changes in the power plan and boost/non-boost state of the CPU will significantly change the outcome. We tried to minimize the variance by

[1]More can be read at: https://learn.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps#hardware-timer-info.

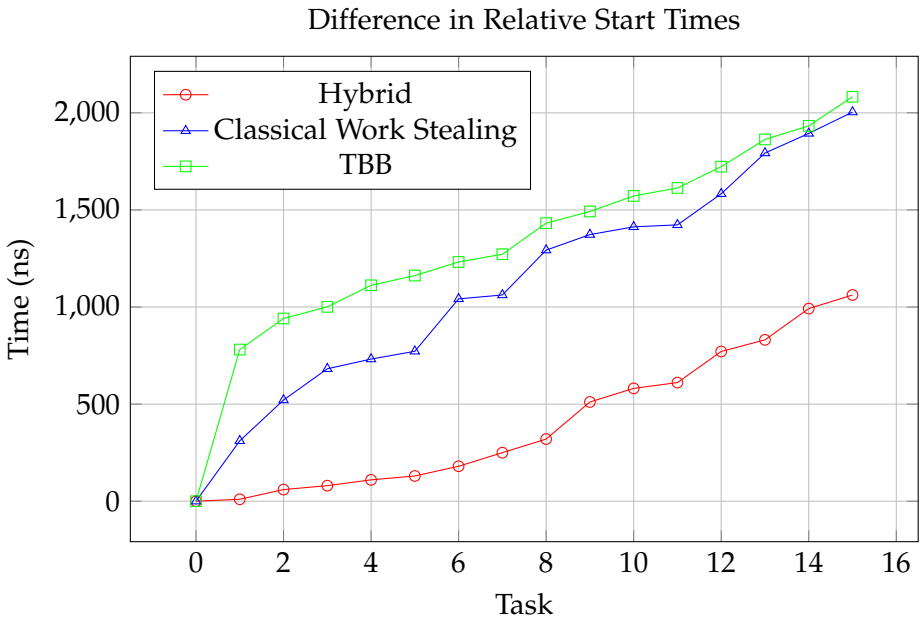


Figure 4.3.: Differences in Start Times from Origin Task on AMD16 System

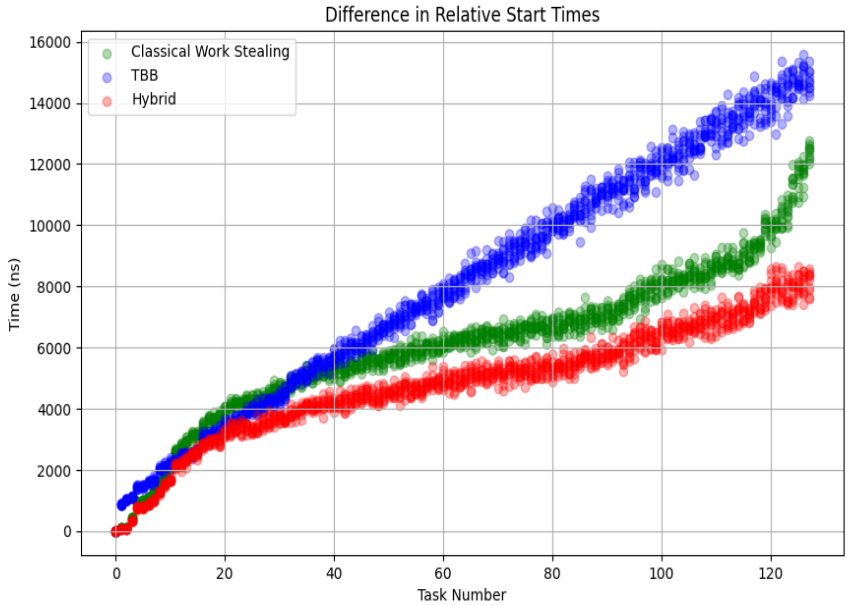


Figure 4.4.: Differences in Start Times from Origin Task on Cloud128 System

running this test 100 times. However, between boots, we got varying median values. Trends shown in the graphs were still similar. Therefore, we are confident in presenting concrete values.

The relative task start times, shown in figure 4.3, display two different paradigms. Work stealing variants `TBB` and `Work Stealing` are displaying clearly start latency inherent to the work stealing. In contrast, our hybrid variant shows little latency for start startup, only breaking the 1000 ns (1 microsecond) at task 14. This figure shows the improvements in intra-task latencies, which resulted in better overall performance. Compared to absolute start latency, we observe a similar picture. Work-stealing variants are slower than the hybrid implementation. While they are similar, our hybrid version is faster beyond run-to-run variation. The time to schedule is under a millisecond for all methods, which means our scheduler can be used for almost all low-latency applications.

Running the same test on a many-core system shows a similar trend. In this case, we can observe the latency during the execution of the first and last few tasks more clearly. TBB implementation showed surprisingly linear times; we suspect it is because of the task pooling (arena) used internally in addition to task streams. In the source files of task streams, similar concerns of linear runtime are raised.

The classical work stealing performed as expected, almost following a logarithmic curve and spiking at the end. This effect is exaggerated when the number of works falls under the threshold for the coupon collector's problem. $log_2(\#of\ cores)$ is in this case 7. From the 120th task onwards, the latency has increased substantialy. This effect is not observed with the mailboxing approach. While Mailboxing and Classical Work Stealing share a similar curvature, mailbox performs better in the beginning, in the middle, and especially at the end. Mailboxing delivers low latency scheduling on many-core CPUs.

To conclude, our improvements to the work-stealing deliver equal (if not better in some cases) performance while delivering lower scheduling latency on top. This section looked at the broad performance metrics; in the following sections, we will dissect the hybrid scheduler into its pieces and discuss their impact on performance.

4.3. Deque Comparison

The performance of the containers holding the tasks directly influences the overall throughput of the system. We will discuss the performance metrics of different de-

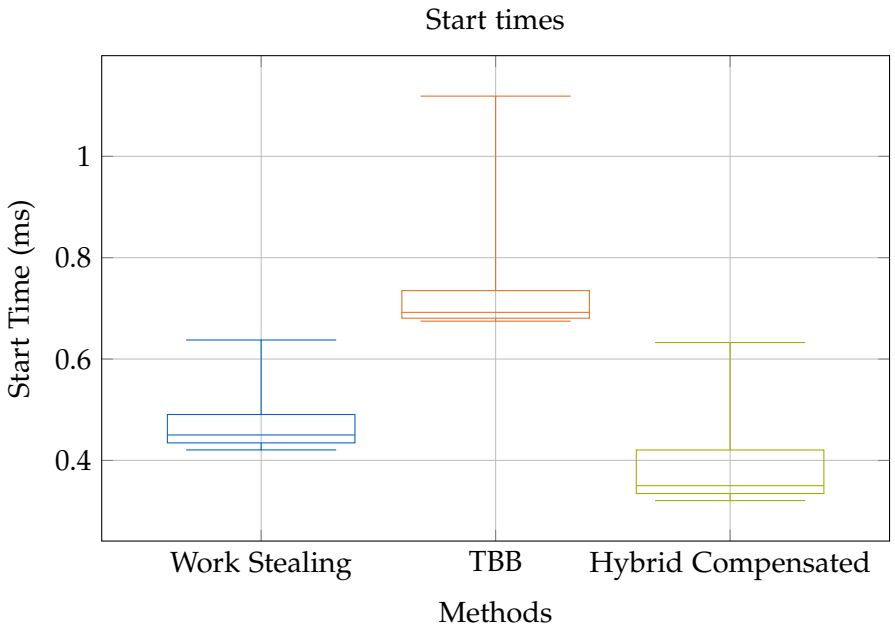


Figure 4.5.: Boxplot comparing Start Times across different Methods

ques in our architecture. More specifically, cache performance and synchronization characteristics. Our test setups are the same as in the chapter above. For recording cache performance, profilers `perf` and `valgrind` are used. Overall, CAS and fence operations are counted manually in the scheduler. We also collected the number of deque operations. The deques will be evaluated using classical work stealing. This is necessary, as we are shifting the complexity of stealing and some local evaluation from deques to mailboxes. A complete system analysis of our hybrid implementation will be presented as well.

4.3.1. Cache

Cache performance is evaluated in two metrics. The cache locality metric can be defined by the `last level cache (LLC)`. Another metric is the cache performance of the L1 cache. Specifically, the `L1 data cache` is important, as `L1 instruction cache` is rarely the performance bottleneck. Frequent load misses in the L1 cache indicate inefficient usage of the localized data, and the same is true for the LLC misses. LLC cache misses indicate how often data is accessed beyond the localized manner. Going from L1 to LLC (L3 in our case) will incur a penalty of around 30 cycles of latency; however, going

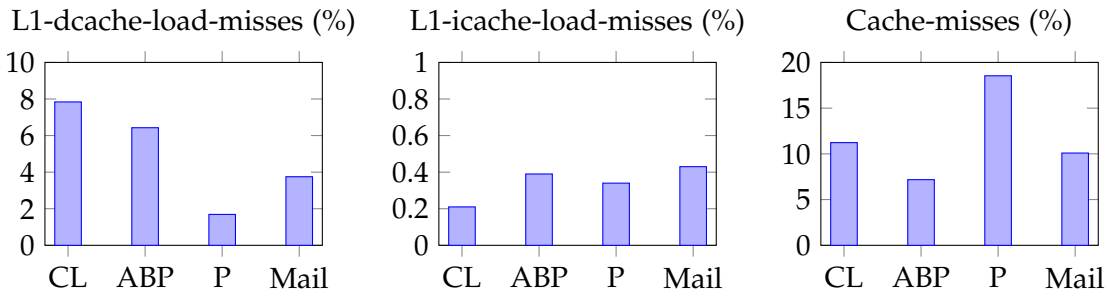


Figure 4.6.: Comparison of Cache Performance Metrics for Deque Variants using Work Stealing. P for private, Mail for Mailboxing, CL for Chase-Lev

from LLC to main memory will be much more costly at 100+ cycles of latency[2]. It is important to remember that there will be a degree of cache misses in every program; however, comparatively, we can evaluate the efficiency of our deques.

The Chase-Lev deque, with an L1-dcache load miss rate of 7.18%, shows a higher degree of inefficiency when accessing L1 data compared to ABP. This can be attributed to the intrinsic structure of the Chase-Lev deque. As discussed in the third chapter, the Chase-Lev algorithm needs to resolve the value of the top in each operation. Considering work stealing is almost 35% of the execution time[3], inherent cache inefficiency of stealing is amplified. In comparison, the ABP algorithm only missed 6.43% of the L1-dcache loads. Frequent access to one more field yielded worse cache performance. As expected, private deque performed the best in L1-dcache, as almost all accesses are localized to the core. At 1.67% private deque, uses the `already loaded` cache the most efficient.

Looking at the broader picture, however, private deque has worse overall cache performance than ABP and Chase-Lev. This is evident from the overall cache performance recorded. The perf event `'cache-misses'` is a combination of LLC-load-misses, LLC-store-misses, and prefetching misses[4]. Because of the explicit communication required during the stealing procedure, the LLC miss rate is higher at 18.54%. Chase-Lev fared a little better at 11% and 7% for ABP, respectively. We observe a trade-off between, better L1 usage versus LLC usage. There is a small performance gain by using private deques in `classical work stealing` at certain tasks with low occurrence of stealing; however, this performance gain is negligible. The original ABP algorithm is more consistent,

[2]Discussed in background.
[3]Measured by AMD's uProf on the AMD16 system
[4]As defined in the source code of perf.

presenting a middle ground. In cases where ABP would overflow, Chase-Lev presents an alternative. While it has the worst cache performance overall, it is the only dynamic deque.

If we include mailboxing in the mix, we will observe another trend. Better L1-dcache usage than ABP and Chase-Lev at 3.75% miss rate. This is close to private deques with less than 2% difference. However, the cache-misses event of mailboxing is much better than private deques. Mailboxing delivers good overall cache performance, substantiating cache awareness. We also observed improvements through using morsels. Executing the same tasks without morsels using TBB yielded approximately 9.6% L1 cache misses and 22% LLC cache misses. Overall, it delivered the worst cache performance in the lineup. Using morsels changed this trend 180, putting TBB's cache performance within 2% of mailboxing. Morsel-based memory usage is a powerful tool for improving cache performance.

4.3.2. Fences and CAS

We continue our investigation of the deques with synchronization operations. Only considering the classical work stealing again, we see a similar trend in which ABP and Chase-Lev display similar metrics. Memory fences and CAS operations are more prevalent in ABP and Chase-Lev, as seen in the graph. Inversely, the private deque has a minimal fence and CAS operations. Avoiding the synchronization cost results in more overhead when stealing explicit communication and memory fences. Private deques are not the silver bullet to solve cache problems of work stealing. However, they are a step forward in memory-aware scheduling.

The ABP algorithm is outperforming the Chase-Lev in the number of operations, as expected. The proportion of CAS and fence operations does not directly translate to the proportion of cache misses. Memory fences are more expensive than CAS operations. Semantically, they propose the same memory barriers, but the `mfence` enforces stricter memory constraints than the `CAS` according to the Intel Manual. While there are multiple `thread_fences` in the C++ standard, only `seq_cst` is guaranteed to use a memory fence. Other types are going to be used for compiler-level operation reordering. Every third fence in Chase-Lev is `memory_release`. It is still a fence operation; however, it is a lot less costly than the textttseq_cst. This explains why ABP is not many-fold faster than the Chase-Lev, given the differences in the number of fences. Again, the ABP algorithm is a good middle ground while using classical work stealing. Using Mailboxing reduces the number of memory fences, but increases the number of CAS

[5]This trend is apparent in other benchmarks as well.

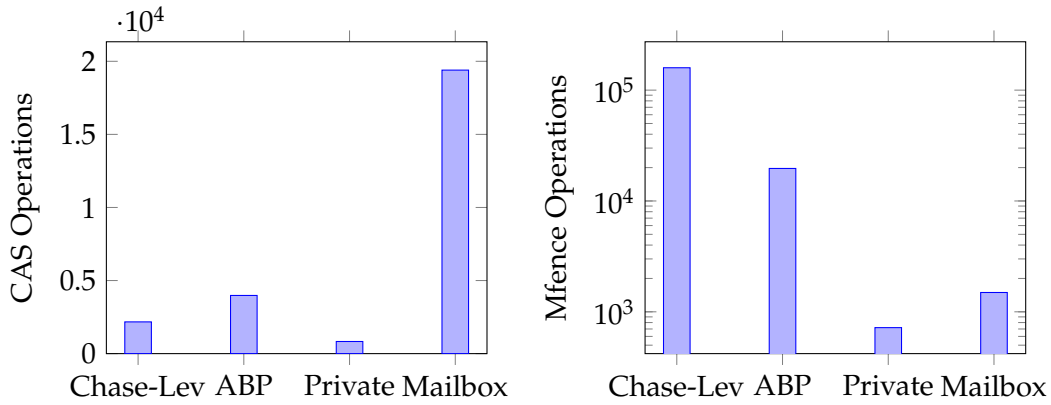


Figure 4.7.: CAS and Fence operations overview for matrix multiplication[5].

operations. We can observe a reduction of fences by 10 times but an increase of CAS operations by 15-20 times. With fewer memory fences, execution can be reordered more, thus making it more efficient as a whole. However, CAS operation is not cheap in high contention situations [23, 33]. We counted the CAS operations of the deque and the mailbox separately. We can conclude that the mailboxing approach actually reduces CAS operations on the shared and contended deques. Increased CAS operations are seen during the insertion to/ popping from the mailbox, which is not highly contended. Another point to consider is the architecture of our scheduler. We are scheduling the tasks twice, which means that in order to finish the execution, all of the tasks need to be marked complete. This is important for the elimination of the task proxies (already executed) in the deques. Thus, the hybrid scheduler uses, by default, more CAS operations. This final step does not hurt the performance of the scheduler, but it explains the discrepancy of the fact that it is using more CAS operations. These results confirm our findings in the performance and cache evaluation: mailboxing is more efficient than classical work stealing. It is important to note that the type of deque used alongside mailboxing did not matter with respect to performance both in raw runtime and in synchronization operations as long as there was no risk of overflowing. This observation makes sense, as the mailbox replaces a lot of operations that would otherwise be executed by the deques.

4.4. Overhead of Mailboxing

To drive our message home, we also calculated the overhead of using mailboxing functions. Most prominently, `pop` and `extractTask` methods are tested. The performance critical point for both methods is the CAS operation. CAS operations become more

expensive with more contention, as they are locking the cache. A normal store can be done in one cycle, yet with CAS; it is 20 cycles latency [24]. Under low contention, typical for the architecture of our hybrid scheduler, the cost of a pop operation in the mailbox remains minimal. The operation primarily involves atomic loads with acquire semantics, simple comparisons, and occasional CAS operations. With cache accesses generally confined to the L1 or L2 cache, the estimated cost is around 50-100 ns. Our simulations of load balancing using mailboxing affirm this estimate; even at 1 consumer and 15 producers (maximum contention on the mailbox) acting on a single mailbox did not cause more than 100 ns latency. Compared to deques, mailboxing is robust and faster, as seen in the figure 4.8. Even though the mailbox operations are twice as fast as the deque operations, trying to use the mailbox multiple times before stealing did not improve the runtime or the latency.

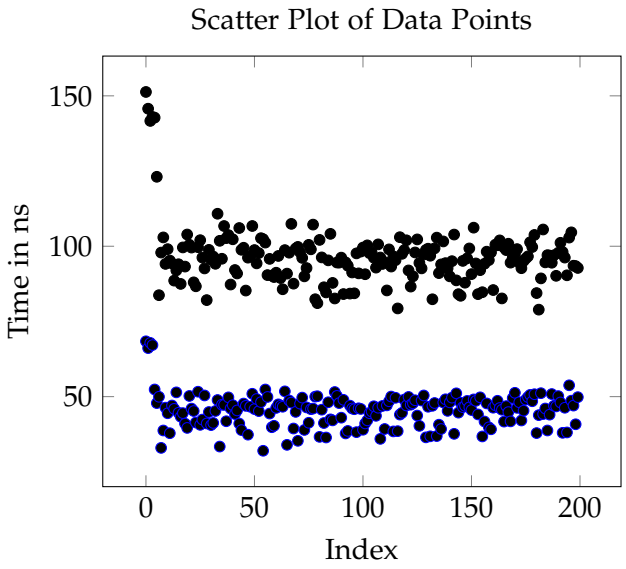


Figure 4.8.: Scatter Plot of Task Retrieval Times, Mailbox in blue, Deque in black

Explicit memory fencing is avoided, and atomic operations contribute negligibly due to limited contention, minimizing delays from cache coherency traffic. The lightweight nature of the operation in such scenarios ensures efficient performance with minimal overhead, allowing the scheduler to maintain scalability and responsiveness. This makes mailboxing well-suited for multithreaded environments with low contention, which is characteristic of well-distributed workloads. If the workload is highly contended, then mailboxing can outshine work stealing by its dual scheduling nature.

5. Future Work

5.1. Memory

Our implementation relies on efficient memory allocation, as per task creation an allocation is called. Naive implementation with `new/delete` proved inefficient for our low latency scheduling goals. Stalling while waiting for allocation is not only affecting the executing core, but also the future execution. This cascading collection of small latencies result in confusing results. Hiding the cost of allocation is a well researched area, commonly done by allocating a large chunk and managing the pointers on that memory. Developing custom memory allocator will definately be beneficial. This is evident from the fact that by changing from `new/free` to TBB scalable allocator, we have reduced the latency by 10% in the allocation phase. Preliminary testing with B+ Trees shows further reduction is possible.

While allocating memory, we can also directly localize the task data to the core. We can design our scheduler logic to favor objects with designated locality. Implementing such a wrapper class is not trivial, however with better coordination in the L4 and L2 cache we can improve the overall performance. If this improvement overcomes the extra work related to it, is an open question. Another improvement can be during the stealing procedure. As discussed in the performance chapter, we can reduce latency and cache related latency with NUMA aware victim selection. However, this is not an automatic process in the current implementation. Making the victim selection biased for already localized data, and with respect to NUMA nodes, we can reduce the LLC cache misses considerably.

5.2. Task Migration

In the current implementation, we are only stealing one task per steal operation. In the literature there are other stealing schemes as well. `Steal Half` and `Task Coelescence`, `continue style stealing`, and `at-least-once semantics` are some examples. The current benchmarks lack IO interruptions or explicit waiting for external data. With continue style stealing we can proactively steal and limit stalling. Hybrid scheduler

already necessitates two times access to the task, which can be leveraged by relaxed semantics of `at-least-once`. This way we can mail the task more than once, while confident this won't cause more overhead (cf. figure 4.8). This excentuates the need for less frequent stealing. Two different solution paradigms can be observed. `Task Coelescence (TC)` makes the number of tasks smaller by reducing the DAG into logical equivalents, combining smaller pieces into larger chunks. `Steal Half (SH)` reduces the number of steals by stealing bigger pieces at once. Both of the solutions can work, however there are overheads related as well. TC needs more information about the workload to work optimally, becoming especially expensive in irregular problems. SH can cause multiple migrations for a group of tasks, thus causing more cache-misses. In optimal conditions (steady and predictable task steams), both can improve the cache performance.

5.3. Parallel Programming

In future work, extending our implementation to support advanced parallel programming techniques can provide additional performance improvements. Leveraging parallel pipelines, parallel sort, and parallel reduce operations can optimize task execution for common patterns, reducing the overhead in task scheduling and improving throughput. Furthermore, utilizing flow graphs to preserve task dependencies would allow for more efficient execution in complex scenarios, such as irregular workloads or task chains that require guarenteed precedence conditions.

Inter-thread variables like `tbb::combinable` offer an efficient way to handle shared data, reducing contention and avoiding unnecessary synchronization points. These mechanisms, along with a better task distribution strategy, can help mitigate bottlenecks in highly parallel workloads, achieving better load balancing and reducing the cost of task synchronization. By incorporating these techniques, the system could further optimize performance for both regular and irregular task patterns.

5.4. Complexity Proof

A formal complexity proof is crucial to establish theoretical bounds, validate performance claims, and compare the hybrid approach with classical work-stealing algorithms. This analysis would provide rigorous foundations for the scheduler's efficiency and scalability claims.

6. Conclusion

This thesis presented a new approach to low-latency scheduling for many-core CPUs, addressing key challenges in work distribution and task management. We started with the weaknesses in the classical work stealing and, step by step, eliminated them. By combining the strengths of work stealing with a mailboxing system, we developed a hybrid scheduler that demonstrates improved performance and reduced latency compared to classical work stealing.

Further improvements to the work stealing part of our scheduler have been presented. Concurrent data structures, such as the ABP algorithm and its dynamic version, the Chase-Lev algorithm, are presented in detail. Another paradigm in concurrent deques, split deques, is also presented, with more detail in the appendices. In the cache analysis, we have observed that the L1 and LLC cache behavior varied between deques. While the split deque performed the best in the L1 cache, it is also the worst in LLC cache misses. The ABP performed better than the Chase-Lev, signaling its good all-rounder performance. The mailbox also delivers good cache performance, having better L1 performance than the ABP and better LLC than split deque. While the type of the deque used next to mailboxing did not matter much, for some workloads ABP deque can overflow. The mailboxing approach with morsels exhibits improved cache performance compared to classical work stealing, substantiating our cache claims.

Performance testing across a variety of benchmarks demonstrates that our hybrid approach consistently matches or outperforms both classical work stealing and Intel's TBB. Notably, our scheduler exhibits significantly lower start-up latency and improved intra-task latencies, translating to better overall performance in both consumer and many-core cloud environments. While our implementation shows improvements, there remain areas for future enhancement. These include further optimizations in memory management to offset the allocation costs, automatic NUMA-aware scheduling, and advanced task migration strategies (e.g., share-half and task coalescence). Additionally, integrating more sophisticated parallel programming constructs, such as parallel pipelines or flow graphs, may extend the scheduler's capabilities, enabling more efficient solutions for more complex problems.

In conclusion, this work contributes to the field of parallel computing by demonstrating a practical approach to low-latency scheduling that scales effectively to many-core architectures. The hybrid scheduler's design principles and performance characteristics offer valuable insights for future developments in high-performance computing systems.

Abbreviations

List of Figures

List of Tables

Bibliography

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. "The Data Locality of Work Stealing." In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '00. Bar Harbor, Maine, USA: ACM, 2000, pp. 1–12. ISBN: 1-58113-185-2. DOI: 10.1145/341800.341801.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. "Scheduling Parallel Programs by Work Stealing with Private Deques." In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Shenzhen, China: ACM, 2013, pp. 219–228. DOI: 10.1145/2442516.2442539.

[3] Advanced Micro Devices, Inc. *AMD EPYC™ 7003 Series Processors*. https://www.amd.com/en/processors/epyc-7003-series. Accessed: 2024-09-15. 2021.

[4] V. Alessandrini. *Shared Memory Application Programming: Concepts and Strategies in Multicore Application Programming*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 012803761X.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors." In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277678.

[6] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. "Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 487–498. ISBN: 9781450304900. DOI: 10.1145/1926385.1926442.

[7] A. Baumstark and C. Pohl. "Lock-free Data Structures for Data Stream Processing." In: *Datenbank-Spektrum* 19.3 (2019), pp. 209–218. DOI: 10.1007/s13222-019-00329-4.

[8] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. "A case for NUMA-aware contention management on multicore systems." In: *Proceedings of the 2011 USENIX Annual Technical Conference*. 2011, pp. 557–562.

[9] R. D. Blumofe and C. E. Leiserson. "Scheduling multithreaded computations by work stealing." In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[10] M. Bohr. "The evolution of scaling from the homogeneous era to the heterogeneous era." In: *2011 international electron devices meeting*. IEEE. 2011, pp. 1–1.

[11] S. Borkar. "Thousand core chips: a technology perspective." In: *Proceedings of the 44th annual Design Automation Conference* (2007), pp. 746–749.

[12] B. B. Brandenburg. "Scheduling and locking in multiprocessor real-time operating systems." In: *Proceedings of the 33rd IEEE Real-Time Systems Symposium*. IEEE. 2011, pp. 29–38.

[13] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[14] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language." In: *International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing." In: *ACM SIGPLAN Notices* 40.10 (2005), pp. 519–538.

[16] D. Chase and Y. Lev. "Dynamic Circular Work-Stealing Deque." In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: ACM, 2005, pp. 21–28. ISBN: 1-58113-986-1. DOI: 10.1145/1073970.1073974.

[17] L. Clarke, I. Glendinning, and R. Hempel. "The MPI Message Passing Interface Standard." In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by K. M. Decker and R. M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[18] R. Custódio, H. Paulino, and G. Rito. "Efficient Synchronization-Light Work Stealing." en. In: *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. Orlando FL USA: ACM, June 2023, pp. 39–49. ISBN: 978-1-4503-9545-8. DOI: 10.1145/3558481.3591099.

[19] L. Dagum and R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.

[20] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. "Scalable work stealing." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–11.

[21] V. Eijkhout. *Introduction to High Performance Scientific Computing*. 2010.

[22] D. Ferraro, L. Palazzi, F. Gavioli, M. Guzzinati, A. Bernardi, B. Rouxel, P. Burgio, and M. Solieri. "Time-sensitive autonomous architectures." In: *Real-Time Systems* 59.4 (Dec. 2023), pp. 568–608. ISSN: 1573-1383. DOI: 10.1007/s11241-023-09404-2.

[23] A. Fog. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*. Tech. rep. Accessed: 2024-09-10. Copenhagen University College of Engineering, 2024.

[24] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. Tech. rep. Accessed: 2024-09-10. Copenhagen University College of Engineering, 2024.

[25] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201575949.

[26] M. Frigo, C. E. Leiserson, and K. H. Randall. "The implementation of the Cilk-5 multithreaded language." In: *ACM Sigplan Notices* 33.5 (1998), pp. 212–223.

[27] D. Geer. "Chip makers turn to multicore processors." In: *Computer* 38.5 (2005), pp. 11–13.

[28] J. R. Goodman and H. Hum. "MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects (2004)." In: 2004.

[29] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. 2nd. Boston, MA, USA: Addison-Wesley, 2003. ISBN: 0201648652.

[30] D. Hendler, N. Shavit, and L. Yerushalmi. "A scalable lock-free stack algorithm." In: *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. 2004, pp. 206–215.

[31] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[32] Intel Corporation. *Intel oneAPI Threading Building Blocks*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html. Accessed: 2024-09-10. 2023.

[33] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Version 80.

[34] A. Kogan and E. Petrank. "A methodology for creating fast wait-free data structures." In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 2012, pp. 141–150.

[35] C. Lameter. "An overview of non-uniform memory access." In: *Commun. ACM* 56.9 (Sept. 2013), pp. 59–54. ISSN: 0001-0782. DOI: 10.1145/2500468.2500477.

[36] Lawrence Livermore National Laboratory. *Introduction to Parallel Computing Tutorial*. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. Accessed: 2024-9-10.

[37] V. Leis, P. Boncz, A. Kemper, and T. Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM. 2014, pp. 743–754.

[38] B. Lepers, V. Quema, and A. Fedorova. "Thread and memory placement on NUMA systems: asymmetry matters." In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 277–289.

[39] T. Liu, C. Curtsinger, and E. D. Berger. "Dthreads: Efficient deterministic multithreading." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 327–336.

[40] A. Malakhov and E. Fiksman. *Pushing the Limits of Work Stealing*. Tech. rep. Accessed: 2024-9-10. Intel Corporation, 2014.

[41] NVIDIA Corporation. *NVIDIA CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/. Accessed: 2024-09-10. 2023.

[42] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins. "Scheduling task parallelism on multi-socket multicore systems." In: *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*. 2011, pp. 49–56.

[43] Oracle. *Java Concurrency API*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html. Accessed: 2024-09-10. 2023.

[44] J. Preshing. *Weak vs. Strong Memory Models*. https://preshing.com/20120930/weak-vs-strong-memory-models/. Accessed: 2024-09-10. Sept. 2012.

[45] G. Rito and H. Paulino. "Scheduling computations with provably low synchronization overheads." en. In: *Journal of Scheduling* 25.1 (Feb. 2022), pp. 107–124. ISSN: 1094-6136, 1099-1425. DOI: 10.1007/s10951-021-00706-6.

[46] D. A. Schneider. "Low-latency trading." In: *Journal of Trading* 6.2 (2011), pp. 10–15.

[47] E. Seo, J. Jeong, S. Park, and J. Lee. "Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors." In: *IEEE Transactions on Parallel and Distributed Systems* 19.11 (2008), pp. 1540–1552. DOI: 10.1109/TPDS.2008.104.

[48] H. Sutter. "The free lunch is over: A fundamental turn toward concurrency in software." In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[49] S. Zhuravlev, S. Blagodurov, and A. Fedorova. "Survey of scheduling techniques for addressing shared resources in multicore processors." In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–28.

Appendices

A. Split Deque Addendum

This chapter presents the split deque implementation details and changes to the scheduler.

First, the classical work-stealing algorithm presented in 1 needs to be modified to accommodate the two-tiered stealing procedure. First, the worker thread must be notified to expose one (or half, depending on the implementation) task. For asynchronous communication, we employ a flag called `targeted`. At each iteration of the worker's loop, the worker exposes tasks if the targeted flag is set.

The assigned is the current work. If it is a valid work, the worker processes the work until it is executable sequentially. In this example, the partitioning is again in halves. If there is no more work in the private region (characterized by the `RACE` value), the worker tries to pop from the public part. Ultimately becoming a thief. Thieves can only target the victims and cannot steal if they haven't put work in the public sector previously. With this change, the scheduler becomes more reliant on the frequency of the steal and the efficiency of the offering work.

Algorithm 9 Low-Cost Work Stealing Algorithm[45]

```
1:  procedure SCHEDULER
2:      while computation not terminated do
3:          if self.targeted then
4:              self.spdeque.updateBottom()
5:              self.targeted ← false
6:          if validNode(assigned) then
7:              enabled ← execute(assigned)
8:              if length(enabled) > 0 then
9:                  assigned ← enabled[0]
10:                 self.spdeque.push(enabled[1])
11:             else
12:                 assigned ← self.spdeque.pop()
13:                 if assigned = race then
14:                     assigned
    ← self.spdeque.popBottom()
15:         else
16:             self.steal()
17: procedure STEAL
18:     victim ← randomThreadID
19:     assigned ← victim.spdeque.popTop()
20:     if assigned = empty then
21:         victim.targeted ← true
22: function VALIDNODE(node)
23:     return node ≠ empty
    and node ≠ abort
    and node ≠ none
```

The split deque needs to manage two bottom indices because of the two-tiered system. Inserting to the private region increases the private bottom index while popping from that region decrements it. If the private region is empty, indicated by line 11, the queue returns a special value RACE. Migrating work from private to public region is done by incrementing the official bottom index, as it points to the end of the public part, essentially exposing a single work. This, too, can be done without fences, as a single fetchAdd instruction on the integer is lock-free.
The public operations remain mostly similar to the ABP algorithm, popTop being entirely the same, working with the age construct and the public bottom index. PopBottom, however, needs to be more complex to compensate the double bottom indices. If there is more than one work in the public region, this operation is relatively cheap. However, if there is a single work (as in most of the cases in this single work exposure scheduler), the queue needs to be reset. This is done by two expensive thread fences in our implementation. This is a conscious design choice, as the popTop can be free of thread fences. This is beneficial, as the ratio of stealing to self-public region access favors this implementation.

Algorithm 10 Split Deque Implementation[45]

```
 1: privateBottom ← 0                                   ▷ Private field
 2: entries ← {}      ▷ Private read-write, public read-only
 3: officialBottom ← 0          ▷ Private read-write, public
    read-only
 4: age ← {0, 0}                                        ▷ Public field
 5: procedure PUSH(node)
 6:     pBot ← self.privateBottom
 7:     self.entries[pBot] ← node
 8:     self.privateBottom ← pBot + 1
 9: procedure POP
10:     pBot ← self.privateBottom
11:     if pBot = self.officialBottom then
12:         return RACE
13:     pBot ← pBot - 1
14:     node ← self.entries[pBot]
15:     self.privateBottom ← pBot
16:     return node
17: procedure POPTOP
18:     oldAge ← self.age
19:     oldBottom ← self.officialBottom
20:     if oldBottom ≤ oldAge.top then
21:         return EMPTY
22:     node ← self.entries[oldAge.top]
23:     newAge ← oldAge
24:     newAge.top ← newAge.top + 1
25:     if CAS(age, oldAge, newAge) = success then
26:         return node
27:     return ABORT
28: procedure UPDATEBOTTOM
29:     pBot ← self.privateBottom
30:     oBot ← self.officialBottom
31:     if pBot > oBot then
32:         oBot ← oBot + 1
33:     self.officialBottom ← oBot
34: procedure POPBOTTOM
35:     oBot ← self.officialBottom
36:     if oBot = 0 then
37:         return empty
38:     oBot ← oBot - 1
39:     self.officialBottom ← oBot
40:     node ← self.entries[oBot]
41:     oldAge ← age
42:     if oBot > oldAge.top then
43:         return node
44:     self.officialBottom ← 0
45:     self.privateBottom ← 0
46:     newAge.top ← 0
47:     newAge.tag ← oldAge.tag + 1
48:     if oBot = oldAge.top then
49:         if CAS(age, oldAge, newAge) = success then
50:             return node
51:     self.age ← newAge
52:     return EMPTY
```