Universiteit Leiden

The effect of Work-Stealing approaches on the runtime of Velvet-Parallelized programs

Geerke van 't Verlaat (s3692396)

Supervisors:
Ben van Werkhoven & Badia Liokouras

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 23, 2026

Abstract

Parallel execution introduces the challenge of load balancing: work must be distributed evenly across threads to achieve good run-time performance for large applications. Work-stealing is a common solution, in which idle threads steal work from other threads, preventing any single thread from remaining idle for long. Velvet is an existing system that executes sequential divide-and-conquer applications in a parallel manner, currently supporting only random work-stealing. This work extends Velvet by implementing and evaluating several new work-stealing approaches, comparing them in terms of run-time, hardware performance, and steal efficiency across six benchmark applications. The results show that no newly implemented approach consistently outperforms the original random approach, though individual approaches do outperform it on specific benchmarks and metrics.

Contents

1 Introduction

With the widespread adoption of multi-core systems, an increasing number of programming models have shifted from serial to parallel execution. Although parallelizing serial programs can significantly reduce execution time, it introduces new challenges, particularly in achieving effective load balancing. The goal of load balancing is to ensure that all processors are utilized effectively by distributing the computational workload evenly across the available processors [1]. No processor should remain idle for an extended period, while the overhead introduced by task scheduling should be minimized. Effective load balancing enables optimal resource utilization and maximizes system throughput.

Load balancing can be achieved either statically, where the workload is divided among processors before program execution, or dynamically, where the workload distribution is adjusted during execution. The advantages of static load balancing include simplicity, predictability, and low computational overhead. However, its main limitation is that it is only effective for programs where the workload can be accurately predicted beforehand. Many parallel applications contain irregular workloads, where task execution times vary or cannot be determined at compile time. When static load balancing is applied to such applications, it can result in workload imbalance, causing some processors to remain idle while others continue processing a large number of tasks. Dynamic load balancing addresses this limitation by allowing tasks to be redistributed during execution, but it can introduce additional synchronization and scheduling overhead [2].

One approach to dynamic load balancing is work-stealing, first introduced in 1994 by Blumofe and Leiserson [3]. Since then, numerous work-stealing algorithms have been proposed, each targeting different aspects of parallel execution and load balancing. Under work-stealing, an idle thread does not simply remain idle. Instead, it attempts to steal tasks from other workers to execute. Some work-stealing algorithms aim to exploit the underlying hardware architecture, while others focus on reducing overhead by minimizing the number of tasks that must be stolen [4, 5]. All work-stealing algorithms must balance three competing objectives: preserving locality to improve cache efficiency, minimizing scheduling overhead, and achieving effective workload distribution across processors [6]. The most popular and widely used work-stealing algorithm is randomized work-stealing, because of its simplicity to implement and its good performance.

This thesis introduces and evaluates multiple work-stealing algorithms through their implementation in Velvet, a framework written entirely in safe Rust that enables the parallel execution of serial programs with divide-and-conquer workloads. The algorithms were evaluated based on the number of steal attempts, hardware performance, and runtime, leading to the central research question addressed in this thesis: how do different work-stealing approaches affect the runtime performance of programs parallelized using Velvet? Answering this question required running and analyzing multiple benchmarks, the results of which show that none of the newly introduced approaches consistently reduced runtime across all benchmarks.

The remainder of this thesis is structured as follows. Section 2 provides the necessary background for this thesis. It introduces the Rust programming language, the divide-and-conquer paradigm, and relevant aspects of modern computer architectures, before discussing load balancing techniques and concluding with an overview of Velvet, the framework used throughout this work. Section 3 then discusses different work-scheduling and work-stealing algorithms from the literature. Section 4 explains how these algorithms are implemented in Velvet, and Section 5 evaluates them through experiments. Finally, Section 6 summarizes the findings and discusses directions for future work.

2 Background

This section provides a background in the Rust language (Section 2.1), divide-and-conquer (Section 2.2), two different types of computer architectures (Section 2.3), and the main problem that work-stealing aims to solve: load balancing (Section 2.4). This section also introduces Velvet (Section 2.5), a framework for parallelizing sequential programs, within which the work-stealing approaches presented in Section 3 are implemented.

2.1 Rust

Rust is a programming language based on memory safety. It ensures that safe Rust code cannot cause memory leaks, dangling pointers, or segmentation faults. This safety is achieved by enforcing rules at compile time through a type system that governs ownership, lifetimes, and mutability. In Rust, each value can either be mutable or immutable and has exactly one owner. When the owner goes out of scope, its lifetime ends, and the value is automatically dropped. If a value is moved into a function, the ownership and thus lifetime of that value is also moved to that function. Consequently, after the function returns, the value goes out of scope and can no longer be used by the original owner. This behavior can be avoided through borrowing. A value can be borrowed by variables that do not own it by creating references. At any given time, Rust allows either one mutable reference or any number of immutable references to a value. However, certain safety restrictions can be circumvented through the use of unsafe code. In unsafe Rust code, the programmer can take five actions that they cannot do in safe Rust code: dereference a raw pointer, call an unsafe function or method, access or modify a mutable static variable, implement an unsafe trait, and access fields of unions [7].

To implement data structures in Rust, the programmer must define a structure specifying its fields and their types, and then implement whatever methods are needed to give that structure its intended behavior. When different data structures share the same behavior, the same method name can be invoked across those structures. This is made possible through traits: trait definitions group method signatures together to specify a set of behaviors necessary to accomplish some purpose. When different data structures share a trait, they can be treated uniformly through that trait, allowing generic code to operate over any type that implements it.

Multiple threads can use shared memory to communicate. However, when multiple threads access shared memory, a procedure is needed to handle concurrent access. One commonly used tool to handle concurrency is a mutex. A mutex stands for mutual exclusion. It ensures that only one thread at a time can access the data protected by the mutex. This is achieved through a lock: when a thread wants to access the data inside a mutex, it must first acquire the lock. Once it has finished accessing the data, it must release the lock, allowing other threads to access it. To allow for multiple owners of a mutex that can be shared across multiple threads, the mutex needs to be wrapped in an Arc (Atomic reference counter). Another approach for handling concurrent access is the use of atomic operations. Atomics provide a lock-free alternative by relying on specialized hardware instructions to ensure safe access to shared data [7].

Rust also provides a powerful macro system that enables metaprogramming, allowing programs to generate or transform code based on input data [8]. There are two types of Rust macros: declarative macros and procedural macros. A kind of procedural macro is the attribute proc-macro. This macro takes as input the item to which the macro is attached and gives as output a token

stream that completely replaces the input [9].

2.2 divide-and-conquer

Divide-and-conquer is an algorithm that takes a problem and divides it into smaller subproblems. The dividing and solving of subproblems happens recursively. Each subproblem is divided into even smaller subproblems until a threshold is reached. Those subproblems are then solved and combined to solve the original problem. Thus the divide-and-conquer algorithm has three steps: divide, conquer (solve), and combine. An important part of the divide-and-conquer algorithm is that every subproblem can be solved separately and simultaneously from each other.

2.3 NUMA architecture vs UMA architecture

There are two types of architectures: symmetric multiprocessing (SMP) and nonuniform memory access (NUMA). The SMP architecture makes use of uniform memory access (UMA), every processor has its own cache and shares a bus to the memory. This means that the access time to memory is always the same, no matter which processor is requesting or where the data lies in memory. A disadvantage of SMP is that scaling this architecture results in longer access times. This is because the processors share a memory bus, and when the bus is "full," a processor has to wait idle until it can access the bus. This system is limited to the bandwidth of the bus.

In a Non-Uniform Memory Access (NUMA) architecture, processors are grouped into clusters (nodes), each with its own local memory. Memory access is faster within a cluster than across clusters (remote or foreign memory), as local memory is directly connected to the processor and does not require interconnect communication. The NUMA ratio is duration it takes to access remote memory in comparison to local memory and is calculated with: $R_{NUMA} = \frac{T_{remoteaccess}}{T_{localaccess}}$ [10].

2.4 Load balancing

A system is considered real-time when correct behavior depends not only on the logical correctness of its operations, but also on the time at which those operations are executed. The goal of work-scheduling is to execute processes in real-time and make effective use of the available processing capacity. It solves two problems: on which processor a task should take place (allocation problem) and in what order tasks should be executed (priority problem) [11]. There are generally two types of work-scheduling strategies: work-first and help-first. In a work-first scheduling strategy, when a worker encounters a new task, it immediately starts executing that task and pushes the currently running task back onto the work-queue. In contrast, in a help-first scheduling strategy, when a worker encounters a new task, it places it on the work-queue and continues executing the current task [12].

An extension of work scheduling is work-stealing and work-sharing. Both are techniques used to achieve load balancing. While work-stealing ensures that a worker does not remain idle when its local work-queue is empty by becoming a thief and stealing work from other workers (victims), work-sharing achieves load balancing by pushing newly generated tasks to idle workers when the owner of the task already has a sufficient number of tasks in its work-queue [12].

Task migration incurs communication and scheduling overhead. If this overhead exceeds the expected performance gain, migration is not worthwhile. Overly aggressive scheduling can also

lead to task migration based on minor load imbalances, causing processors to spend more time migrating tasks than executing them, a behavior known as processor thrashing [13].

2.5 Velvet

Velvet [14] is a programming model completely written in safe Rust code that parallelizes sequential divide-and-conquer programs. It uses Rust metaprogramming, using procedural macros, to transform sequential functions tagged with `#[ spawnable ]` into parallel ones. These new parallel functions are executed in a divide-and-conquer manner with random work-stealing.
In Velvet, there are two core operations to execute a divide-and-conquer algorithm provided by the programmer: spawn and sync. The first, spawn, creates tasks that may run in parallel. Variables written to by spawns are marked as sensitive, and before each sensitive variable is accessed, there is a sync operation that awaits until a spawn is completed to ensure correctness.

Each parallel worker maintains a work-queue with tasks that are invocations of the spawnable functions. For each spawnable function, there is a VelvetFrame that represents either a call to the function (the IN variant) or the function's return value (the OUT variant), there is also a STOLEN variant that represents that a task is stolen. When a Velvet application starts, a worker is created for each thread. Each worker is pinned to a CPU core when possible and is associated with a work-queue and a corresponding Stealer. In addition, every worker maintains references to the stealers of all other workers. These stealers are invoked by other workers whenever they attempt to steal work. The worker running on the current thread, also called the root-worker, "begins the top-level computation and, upon reaching a spawnable function, begins spawning work by enqueueing VelvetFrames into its local work-queue." [14] The remaining workers immediately begin the steal-loop.

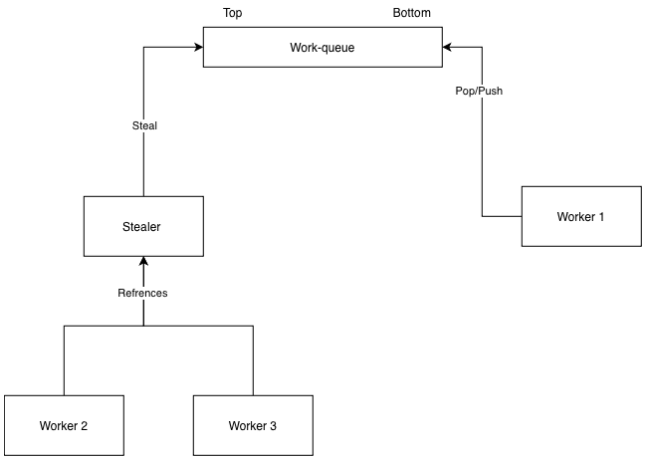

Figure 1: Connections between workers, work-queues, and stealers. Each worker owns a work-queue, and each work-queue has a corresponding stealer that provides access to its tasks. Every worker holds a reference to the stealer of every other worker's work-queue, allowing it to attempt to steal tasks from any other worker.

When a stealer's steal function is invoked, it selects a task from its worker's queue to return to the thief. The selected task is then validated using the appropriate stealing logic. If the validation succeeds, the thief has successfully stolen the task. Otherwise, the steal attempt fails, and the thief must retry by selecting another victim and attempting another steal. When a Stealer successfully steals a task, it replaces the VelvetFrame of the stolen task in the victim's queue with a locked Stolen variant of the VelvetFrame containing None to signal to the victim that a task is stolen. After the stolen task is executed, the thief writes the result, if there is a return value, into an OUT VelvetFrame and releases the lock to signal to the victim that a stolen task is successfully executed.

3 Related Work

This section introduces several work-scheduling and work-stealing approaches. In a lot of papers, there is a very small distinction between work-scheduling and work-stealing. The latter, most of the time, is seen as an extension of the first. The two concepts are closely intertwined, as the way work is scheduled influences how work is subsequently stolen.

3.1 Work-scheduling

This subsection introduces two work-scheduling approaches.

3.1.1 Work pushing

Work pushing [15] is an approach designed to reduce the mismatch between task placement and data locality. The algorithm proactively transfers tasks to workers whose NUMA nodes contain the data required for execution. To achieve this, it relies on data-flow information, specifically requiring knowledge of the data that a task will access prior to its execution. Each worker has a Multi-Producer, Single-Consumer (MPSC) First in First out (FIFO) queue for transferring tasks. The approach consists of two complementary algorithms. The first is responsible for transferring tasks that are ready for execution to a more suitable node when necessary. Specifically, a task is moved from one MPSC FIFO queue to another when it is determined that the data required for its execution resides in a different cluster. The algorithm selects a target worker within that cluster at random. If the destination worker's FIFO queue is full, the transfer is aborted, and the task is reinserted into the originating worker's queue. The second algorithm is responsible for transferring a task from a FIFO queue to a local work-queue. While the FIFO queue is not empty, it transfers the last added task from the FIFO queue to the local work-queue. It transfers the last added task in the FIFO queue because that task had the highest probability that the data still lies in the local memory.

The work pushing approach assumes that a task's data is stored in a single, continuous memory region. This is not always the case and does not optimize for outgoing dependencies. These problems are solved in the enhanced version of work pushing [16]. This version determines whether a newly created task by a node needs to be transferred to a remote node using three heuristics: input-only, output-only, and weights, and three variables: a data array that stores the cumulative size of input and output buffers of a task for each node in the system, the incoming dependencies, and the outgoing dependencies. The algorithm determines, for each dependence, whether it needs to be taken into account, the size of the buffer, and the node containing the buffer. It updates the data array accordingly by multiplying the size of the buffer by the weight associated with the dependence. The assigned weight reflects whether the dependency is incoming or outgoing, accounting for the fact that read and write accesses typically have unequal latencies. For the outgoing dependence, there is a chance that the node is not yet known. If this is the case, the size of the buffer is added to the total size, but the data array is not updated.

After these steps, the algorithm determines whether a task should be pushed to a remote node and, if so, identifies the target node for the transfer. The selected target node is a node that contains at least one of the required data buffers and minimizes the overall data access cost. A task is not pushed to a remote node if the total buffer size is lower than a certain threshold. This

avoids cases in which the overhead of a remote push cannot be compensated by the improvement in the task's execution time.

3.1.2 Locality-Guided

Locality-guided is an approach designed by Acar [17]. It can be viewed as a work-stealing approach, although it more closely resembles a work-scheduling approach. The main idea is that tasks that are working with the same data should be executed by the same worker, ideally by the worker that lies on the same processor as the data. To achieve this, each task is assigned an affinity to a specific worker, and in addition to its regular work-queue, each worker maintains a FIFO queue called a mailbox, which contains tasks that have an affinity for that worker. When a task is created, it is placed in the creator's work-queue, and a copy of the task is inserted into the mailbox of the worker with the highest affinity for it. When a worker's work-queue is empty, it first checks if there are any tasks in its mailbox and executes them if there are, before going over into work-stealing. To prevent duplicate execution, each task is associated with an atomic boolean that indicates whether the task is being processed or has already been completed. A worker sets this flag when it starts executing the task. Any worker that subsequently encounters a flagged task discards it. To guarantee correct task synchronization, each task is assigned a timestamp. Workers compare this timestamp against the current synchronization state and ignore tasks that are deemed outdated.

3.2 Work-stealing

Work-stealing relies on a double-ended work-queue. Tasks executed on the local core are taken from the bottom. Every time a task is made, it is stored at the bottom of the thread's work-queue. When another thread attempts to steal work, it takes a task from the top of the work-queue. These tasks tend to have more sub-tasks and therefore take longer to execute, reducing the total number of steal operations required. This behavior arises from the hierarchical nature of divide-and-conquer algorithms. Larger tasks are encountered and pushed onto the work-queue first. As these tasks are executed, they generate smaller sub-tasks, which are subsequently added to the queue. Consequently, tasks closer to the top of the work-queue are generally larger and contain more remaining work than those near the bottom. The idea is that smaller computational tasks are handled locally by the thread, while larger, more computationally expensive tasks are more likely to be stolen.

3.2.1 Randomized work-stealing Algorithm

In randomized work-stealing (RWS), first introduced by Blumofe and Leiserson [3], the thief randomly chooses a work-queue from another thread (the victim) and steals the topmost task from that queue. A thief fails when the work-queue of the randomly chosen victim is empty.
There are different variants of RWS described by Mitzenmacher [18].

- *Threshold stealing*: Only steal from threads where the number of tasks in the queue is at least a certain size. This has a higher probability that there is a failed steal attempt, but improves the chances that the cost of transferring the task from the victim to the thief is worthwhile.
- *Preemptive stealing*: Instead of only stealing when a thread's work-queue is empty, it begins stealing when the number of tasks left in the work-queue is below a certain amount.

- *Repeated stealing*: When a thief fails, it tries a certain number of times again.
- *Multiple steals*: When a thief steals from the victim, instead of taking one task, it takes multiple tasks. This can reduce the time a task spends in the system when the victim's work-queue is large.

Randomized work-stealing has an execution time of: $T_P = \frac{T_1}{P+O(T_\infty)}$, where T_1 is the minimum execution time of serial execution, T_∞ is the minimum execution time of an infinite number of processors, and P is the number of processors.

Randomized work-stealing leads to good global load balancing, but can have poor data locality. So there can be a mismatch between the task and data placement. There is a high probability that the data of a task is located on a different node than the task itself. Consequently, the input data has to be fetched from a remote side. This leads to high-latency, long-distance data transfers and reduced performance of the application [19].

Another factor contributing to poor data locality, is the high probability of stealing from a worker located in a different cluster in a NUMA architecture. When this occurs, the chances that a processor needs to access data that is located in remote memory, this takes more time than accessing data in local memory.

3.2.2 Affinity-aware work-stealing

Affinity-aware task stealing [4] is a task stealing algorithm based on the underlying topology of hardware. During the initialization of the library, the system topology is translated into a tree structure. The algorithm defines a set of stealing domains corresponding to the clusters present in the architecture, with each domain represented as a child of the root node in the tree. Workers primarily steal tasks from other workers within the same domain. Only when no suitable victims are found locally, falls the thief back to randomized work-stealing.

The algorithm further minimizes overhead by eliminating the need for random victim selection. Each worker maintains a state variable that classifies it as either a thief, a victim, or None. A worker's state is typically victim, it transitions to None when its work-queue falls below a predefined threshold, and to thief once it begins stealing work. During task-stealing attempts, a thief first searches for workers marked as victims within its local stealing domain. This targeted selection strategy reduces the time required to identify a suitable victim and trades the overhead associated with random victim selection for overhead associated with checking which wokres are self-reporeted victims.

An optimization of this work-stealing approach is described by Saman Barghi and Martin Karsten [20]. In this optimization, when a thread fails to steal in its local stealing domain, instead of going over into randomized work-stealing, it tries to steal from other clusters with increasing NUMA distance. This algorithm also keeps track of how many threads are trying to steal from a local node, when there is only one non-empty work-queue (only one potential victim) and many threads are trying to steal from the local node, a thread back-offs. This is to limit the contention over work-queues on the local NUMA node.

3.2.3 Localized work-stealing algorithm

Another variant of work-stealing is steal-back [21], also known as localized work-stealing, where processors preferentially reclaim work that they originally generated. Each processor maintains

ownership of its spawned tasks and tracks the processors currently executing work stolen from it. When a processor steals a task, it records itself as the owner of this task's list. Upon becoming idle, a processor first selects a victim from processors in this list, instead of selecting a random victim. When a steal-back attempt is unsuccessful, the processor removes the victim from its list and selects another. When the list is empty, the thief falls back to general work-stealing. This general work-stealing can be random, or it may use other work-stealing strategies. When a thief steals from a victim that has already stolen work, it records itself in the original owner's list rather than in the victim's list. This variant has an expected runtime of $T_P = \frac{T_1}{P+O(T_\infty P)}$

The intuition behind this variant is that sometimes a processor performs some computations and stores the results in its cache. When trying to steal-back its own work, it can benefit from the fact that the results are stored in its cache.

This variant has two optimizations: hashing and mugging. The first one addresses the problem of pile-up. A pile-up arises when multiple workers are working on one worker's tasks. Then the chances that a thief steals a task belonging to that worker are higher, which increases the number of workers working on the original worker's tasks and makes this issue worse. When a pile-up happens, many workers only focus on one critical path, which leaves the rest of the computation unexplored and limiting how much the overall execution time can actually be reduced. The solution to this problem is for a thief to first randomly select a worker that is still executing its own task. It then randomly selects a victim from among the workers currently executing tasks that were originally stolen from that worker. This helps with the critical path analysis. Now a thief has $\frac{1}{k}(\frac{1}{P_1} + \ldots + \frac{1}{P_k})$, where k is the number of workers working on their own task, and P is the total number of workers, the probability of selecting one of the critical paths.

In mugging, the worker that steals work back does not take only the top-most task. Instead, it takes the entire work-queue, either including or excluding the task currently being executed by the victim. This result that a normal steal can only result in one steal-back. Therefore, the expected number of steal-backs and steals is $O(PT_\infty)$

3.2.4 Deterministic work-stealing

Deterministic work-stealing is an approach proposed by Varisteas and Brorsson [22]. The goal of this approach is to eliminate randomness from the victim-selection process by enforcing a set of deterministic rules. Instead of selecting victims at random, a thief chooses from a predefined set of workers based on the system topology.

The topology used for victim selection does not necessarily have to correspond to the physical hardware topology, it may also be a virtual topology. The approach requires that all workers are pinned to CPU cores and that a metric exists for measuring the communication distance between cores. For example, cores can be modeled as nodes in a two-dimensional grid, where the communication distance between neighboring cores is one. A thief is then restricted to stealing only from workers whose cores lie within a maximum communication distance, such as two, from its own core.

The experiments conducted by the Varisteas and Brorsson demonstrate that this approach reduces stealing overhead, resulting in faster and more uniform task relocation and a reduction in the number of failed steal attempts.

3.2.5 Priority based selection

In a priority-based victim selection, proposed by Nakashima [5], each victim is given a priority based on the number of tasks left in their work-queue. The victim with the largest number of tasks left has the highest priority. In this algorithm, the thief selects k number of victims randomly with $1 \leq k \leq n$ as victim pool, where n is the total number of workers. Out of this victim pool, the thief selects the victim with the highest priority. If the victim pool is as big as the worker pool, then the thief selects the victim with the largest work-queue. The intuition behind this approach is that stealing from a high-priority victim reduces the likelihood of selecting small tasks and increases the likelihood of successfully stealing a task. For this assumption to hold, it must be the case that a large number of remaining tasks in a worker's work-queue implies that the remaining tasks are large.

Based on the experiments conducted by Nakashima, setting (k = n) (priority-all), where the number of potential victims equals the number of workers, resulted in considerable performance improvements. Nakashima also evaluated a configuration with (k = 3) potential victims, but this yielded less significant performance improvements than the priority-all approach.

4 Approach

This section presents five different work-stealing approaches implemented in Velvet. The newly implemented approaches are: VictimRegistery (Section 4.1), mugging (Section 4.2), Localized work-stealing (Section 4.3), Topology-aware work-stealing (Section 4.4), and priority (Section 4.5)[1].

In the original Velvet implementation, a worker provides a get_random function that generates a random number while ensuring that the generated number does not correspond to the worker's own ID. In the newly implemented work-stealing approaches, a worker must also be able to generate a number that is its own ID. Therefore, an additional `get_random_with_self` function is introduced. This function generates a random number without checking whether the generated number is the worker's own ID.

4.1 VictimRegister

One newly implemented approach combines a small optimization introduced in Section 3.2.1, where a thief may only steal from workers that have at least a certain number of tasks remaining in their work-queue. This idea is further developed in Section 3.2.2, where a worker explicitly marks itself as a potential victim once the number of remaining tasks in its work-queue exceeds a predefined threshold. The idea is that there is one global registry in which a worker registers itself once the number of tasks in its work-queue exceeds a given threshold, and unregisters itself once that number falls below that threshold. Thieves then only consider workers currently present in this registry as candidates for stealing.

To implement this mechanism in Velvet, a new data structure called the VictimRegistry is introduced. The registry is shared among all workers and contains a queue protected by a Mutex lock, together with two methods: register and unregister. As their names suggest, register adds a worker identifier to the queue if it is not already present, whereas unregister removes a worker

[1]The implementation is available on: https://github.com/Geerke/thesis

identifier from the queue. Each work queue holds a reference to the shared VictimRegistry, along with an AtomicUsize tracking its current length, an AtomicBool indicating whether its owner is currently registered, and the identifier of the owning worker. Together these let the queue automatically register or unregister its worker in the VictimRegistry whenever the queue length crosses the relevant threshold.

To ensure a worker's registration is updated only when a threshold is actually crossed, each push, pop, or steal obtains the previous value of an AtomicUsize length counter via fetch_add or fetch_sub, and compares it against the threshold. If the threshold is crossed, an AtomicBool is then compared-and-swapped to flip the worker's registration state, and only a successful swap triggers a register/unregister call on the VictimRegistry. This prevents redundant registry updates if multiple threads observe the same crossing concurrently. But if the length update happens outside the queue's lock, a window opens between the actual queue mutation and the counter update meant to reflect it, this can caus missed or incorrect registrations. To prevent this, the length update is performed inside the same lock as the queue mutation, so the counter always reflects the true post-mutation state and can't race against it.

For example, if a worker pushes a task onto its own queue via spawn at the same moment another worker steals a task from it, both operations update the length counter atomically within their respective queue-lock critical sections. This guarantees each operation observes a length consistent with its own mutation, so the resulting VictimRegistry registration state is always correct regardless of how the two operations interleave.

To prevent thrashing, a small gap is maintained between the task threshold for registering a worker and the threshold for unregistering it. This gap ensures that when the number of tasks in a worker's work-queue fluctuates around the threshold, the worker does not repeatedly register and unregister itself in the VictimRegistry. The threshold to register, also called HIGH, is set to six and the threshold to unregister, also called LOW, is set to three.

The generated steal function is modified such that the thief no longer randomly selects a victim from all available workers. Instead, it selects a victim randomly from the workers registered in the VictimRegistry.

4.2 Mugging

A newly implemented approach is mugging, which was first introduced in Section 3.2.1 and further explained in Section 3.2.3 as one of the optimizations of Localized work-stealing. In this approach, when a thief attempts to steal a task from a victim, it steals all tasks from the victim's work-queue instead of only a single task. This is implemented by first determining the length of the victim's work-queue and then iterating over all available tasks. For each task, the corresponding frame is retrieved from the victim, after which the standard stealing logic is applied.

4.3 Localized work-stealing

One approach newly implemented in Velvet is the localized work-stealing approach introduced in Section 3.2.3. In this approach, the thief first attempts to steal-back tasks that were originally stolen from them. To implement this behavior, each Velvet worker requires a stealers queue, in which other workers store their IDs when they steal a task from it, as well as a variable that tracks the owner of the task currently being executed by the worker. Because of the way Velvet

executes programs, where the root worker initially spawns the first task from which new tasks are spawned and the remaining workers must obtain work through stealing, this variable is initially set to None for all workers except the root worker. When a worker successfully steals a task and begins executing it, the variable is updated to the ID of the corresponding victim.

When a thief successfully steals a task from another worker, it first invokes that worker's stealer to determine the owner of the stolen task. Based on this information, it conditionally invokes the appropriate stealer again to add its own ID to the corresponding worker's stealers queue. This additional step is not performed when stealing back work. When a victim later needs to select a target to steal from, it first checks whether its own stealers queue contains any worker IDs. If the queue is not empty, it attempts to steal from one of these workers before falling back to selecting a random victim. If a thief retrieves a victim ID from the stealers queue but the steal attempt is unsuccessful, that victim's ID is removed from the queue. If the steal attempt succeeds, the thief does not add its own ID to the victim's stealers queue.

A worker's stealers queue can be implemented either as a hash set or as a vector-based queue. Both approaches have advantages and disadvantages. When worker IDs are stored in a hash set, no ordering is maintained. When retrieving an ID from the set, a worker is selected randomly. However, a hash set does not allow duplicate values, meaning that no additional check is required before inserting an ID. Therefore, inserting an ID has an average complexity of O(1). A vector-based queue has the advantage of maintaining ordering, allowing a thief to select workers based on stealing history. For example, it can prioritize stealing back from the worker that stole from it first or from the worker that stole from it most recently. When a thief steals back from a worker that most recently stole one of its tasks, the probability that the data required for the stolen task remains in the thief's cache increases. However, a drawback of a vector-based queue is that it allows duplicate entries. To prevent duplicate IDs, the stealer must first check whether a worker's ID is already present in the queue before inserting it. Since this requires a linear search through the queue, the insertion operation can become computationally expensive, especially when the queue grows large or is accessed frequently.

A disadvantage of the localized work-stealing approach is that, in Velvet, the root worker initially owns all original work. Since it spawns the initial tasks that are later stolen by other workers, ownership can become concentrated around the root worker. This creates a bottleneck that hashing cannot alleviate, as there are no other owners to distribute the workers across. To address this issue, workers do not retain the root worker as the owner of stolen work. Instead, when a worker steals a task from the root worker, it becomes the new owner of that task and all subtasks subsequently spawned from it. However, the thief still adds its ID to the root worker's stealers queue, allowing the root worker to steal-back tasks.

Because the root worker initially spawns the first tasks and all other workers must first obtain work through stealing, a randomly selected victim during the first steal attempt has a high probability of still being in the process of stealing work. To prevent this, when a worker attempts to steal for the first time, it first tries to steal from the root worker instead of selecting a random victim.

4.3.1 Optimazations

Localized work-stealing incorporates two optimizations introduced in Section 3.2.3. One of these optimizations, referred to as hashing, modifies the victim selection process. Each worker maintains

the identity of the original owner of the tasks it is currently executing. If a worker is executing tasks that it originally owned, those tasks have not yet been stolen.

When a thief is not stealing back its own work, it first iterates over all workers and collects those executing non-stolen tasks into a vector. From this vector, it randomly selects a target worker. It then identifies all workers currently executing tasks that originated from the selected target and randomly chooses one of these workers as the victim. To reduce overhead, the target worker is selected only once per stealing phase and is not reselected after each unsuccessful steal attempt.

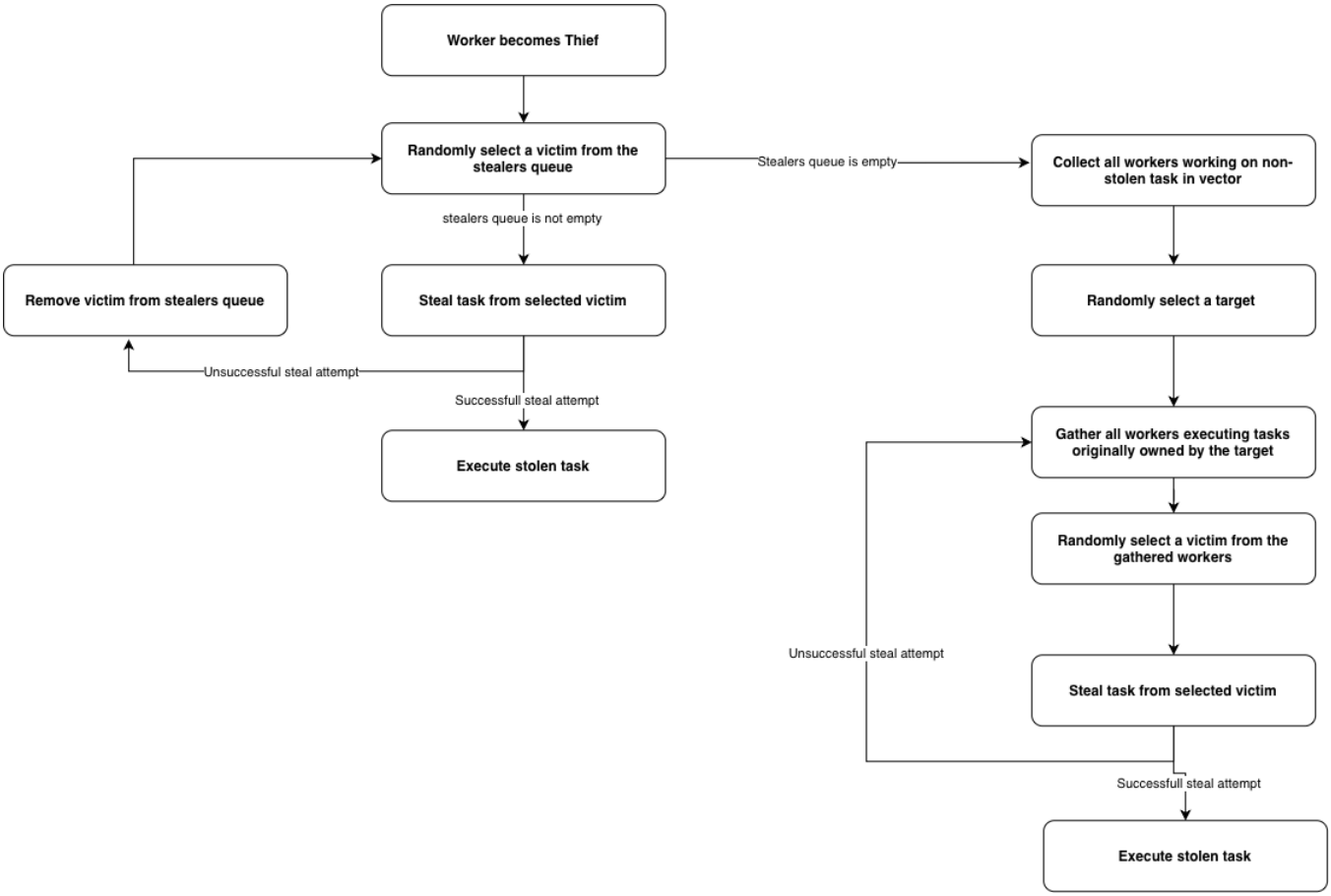

Figure 2: Flowchart of the victim selection process for steal-back hashing. The left side shows the default steal-back process. Once the stealers queue is empty, the process on the right is used instead of randomly choosing a victim: a target worker is selected, and a victim is then chosen from among the workers executing tasks originally owned by that target.

An alternative way to implement this behavior is to use the VictimRegistry introduced in Section 4.1. Every worker currently executing one of its own tasks is registered in the registry. When a thief has no available victims in its stealers queue, it randomly selects a target worker from the registry. The thief then copies the target worker's stealers queue, appends the target worker itself to the copied queue, and randomly selects a victim from the resulting set of candidates.

Another optimization approach is mugging, as explained in Section 4.2. In this optimization, the thief only "mugs" the victim when attempting to steal back work, rather than when performing a regular random steal attempt.

4.4 Topology-aware work-stealing

Another approach implemented in Velvet is the Topology-aware work-stealing approach introduced in Section 3.2.2. In addition to a work-queue and a Stealer, each worker is provided with a hash map that maps every worker ID to its corresponding CPU core ID, as well as a vector containing the CPU IDs of all cores located on the same NUMA node. This vector is obtained using the hwloc2 crate[2]. The stealer includes an additional boolean function that checks whether a worker has enough work to be a valid victim. If the worker's work-queue contains fewer than a given threshold of tasks, it returns false, indicating that the worker is not a suitable victim. Otherwise, it returns true, indicating the worker has enough work to steal from.

During worker initialization, each worker is first assigned to a CPU core. The NUMA node associated with that core is then identified, and the CPU set containing all cores on the same NUMA node is obtained using the hwloc crate. Using the previously established mapping between CPU cores and workers, this CPU set is converted into a vector of worker IDs and stored by the worker. This enables each worker to be aware of the other workers belonging to its local NUMA domain.

For topology-aware work-stealing to function correctly, all workers must be pinned to a specific CPU core. Otherwise, a worker may migrate to a different core, potentially located on another NUMA node. This migration can cause the vector containing the worker IDs of workers on the same NUMA node to become inaccurate. When the worker is not pinned to a core, it falls back into random work-stealing.

During the stealing phase, when a thief attempts to find a victim, it first checks whether the vector containing the IDs of all workers on the same NUMA node is available (i.e., the workers have been successfully pinned to CPU cores). If this vector is available, the thief randomly selects a worker from this vector as a potential victim until a steal attempt succeeds. Before attempting to steal, the thief first verifies whether the potential victim has sufficient available work. If so, the thief proceeds with the steal attempt using the same stealing logic described in Section 2.5. Each time the thief attempts to steal from a worker, that worker's ID is added to a separate vector, referred to as tried. This vector ensures that the thief does not attempt to steal from the same worker more than once. If no task can be stolen from any worker on the same NUMA node, the thief randomly selects a worker from a remote NUMA node, using the tried vector to avoid selecting a worker that has already been attempted.

4.5 Priority

The priority-based victim selection introduced in Section 3.2.5 is implemented for both the random and topology-aware approaches, and requires no architectural modifications. In the random approach, the thief selects the worker with the largest number of remaining tasks in its work-queue as the victim. In the topology-aware approach, the thief performs the same selection but restricts its search to workers on the same NUMA node.

This priority-based victim is determined only once per steal loop, at its start. If the resulting steal attempt is unsuccessful, the algorithm falls back to random victim selection for the remainder of that loop. In the topology-aware approach, this fallback is likewise restricted to workers on the same NUMA node.

[2]https://docs.rs/hwloc2/latest/hwloc2/

5 Experiments

In this section, the performance of the different work-stealing approaches described in Section 4 is evaluated using six benchmark tests based on divide-and-conquer algorithms. For each benchmark, the speedup relative to the random work-stealing approach is measured (Section 5.5), together with the number of successful and unsuccessful steal operations and the time workers spend attempting to steal work under each approach. Additionally, for the steal-back approach, the number of successful steal-back attempts is measured (Section 5.3). For all approaches, the hardware performance is also measured (Section 5.4).

5.1 Setup

The experiments are performed on the Distributed ASCI Supercomputer 6 (DAS-6), which is a x86/AMD-based system equipped with two 16-core AMD EPYC-2 (Rome) 7282 CPU.

All benchmarks are written in safe Rust version 1.89.0 and are written by the original creator of Velvet, Badia Liokouras. They are compiled with the standard rustc compiler in release mode. Parallel speedup is computed relative to the sequential Rust implementation of each benchmark and compared to the speedup of the random stealing approach.

5.2 Benchmark test

As noted above, the benchmark tests used to compare the different work-stealing approaches to the random work-stealing approach are written by the original creator of Velvet. There are six different benchmark tests. The benchmark tests are: Fibonacci (fib), Adaptive integration (adapint), Traveling Salesperson (tsp), Nqueens, Barnes-Hut (bh), and Matrix Multiplication (matmul). The Fibonacci benchmark is mostly used for framework overhead, because every call only performs one integer addition, resulting in minimal computation.

Bench-mark	Problem size	Seq. threshold (s)	Seq. runtime	Nr. of successful steals
fib	n = 50	n=25	50.99	834
adapint	[a, b] = [0,8.0 · 10^6] $\epsilon = 10^{-4}$	diff. 1000	25.02	820
tsp	Nr. cities: 19	Nr. cities: 11	48.31	3,458
nqueens	N = 15	Rem. rows: 10	29.4	1,255
bh	Nr. bodies: 10^6	Nr. bodies: 50	38.97	3,502
matmul	Matrix size:: 4096 x 4096	Matrix size: 64 x 64	25.48	1,117

Table 1: Problem sizes, sequential thresholds and runtime metrics for each benchmark, including the (optimal) sequential runtime, given these inputs. The reported number of stolen jobs are an average when executed on 32 threads.

5.3 Steal-attempts

The evaluation of stolen jobs is based on the average of five runs conducted with 32 threads. The analysis considers both the total number of steal attempts, the number of successful steal attempts, and for the steal-back approaches the number of successful steal-backs (Table 2). In the two mugging approaches, every individual attempt to reclaim a task is counted toward the total number of steal attempts and successful steal attempts, even if multiple tasks are reclaimed during a single steal operation. However, in the steal-back mugging approach, when reporting the number of successful steal-back attempts, only the first reclaimed task is counted, so each steal-back operation contributes at most one successful steal-back attempt.

Furthermore, the average time a worker spends attempting to steal a job is measured (Table 3). For the two mugging approaches, only the time required to reclaim the first task is included in this measurement.

Table 2 shows the total number of steal attempts and successful steal attempts for each approach and benchmark. This provides partial insight into the stealing overhead shown in Table 3, though it does not fully explain it, as the cost of individual steal attempts also plays a role. The approach with the highest steal success rate for each benchmark is highlighted in bold.

The table shows that the random approach performs a large number of steal attempts, but only a small fraction of these attempts are successful, especially for benchmarks with large workloads such as *bh* and *matmul*. Almost all newly introduced approaches reduce the number of steal attempts compared to the random approach. The most notable exception is the hashing approach, which not only fails to reduce the number of steal attempts but also achieves a substantially lower steal success rate.

For the steal-back approach, steal-backs account for approximately 40% of successful steals across the evaluated benchmarks. The Steal-back Mugging approach does not reduce the number of steal attempts for the smaller benchmarks. However, for the larger benchmarks, it significantly reduces the number of steal attempts while also increasing the fraction of successful steal attempts. Both steal-back optimizations reduce the number of steal-backs, but the mugging approach achieves a higher steal success rate than the standard steal-back approach. Interestingly, the Random Mugging approach achieves a substantially higher steal success rate than the Steal-back Mugging approach. A possible explanation is that Random Mugging always mugs the selected victim, meaning that multiple consecutive steals from the same victim can increase the likelihood of successfully stealing work. Moreover, a thief only attempts to steal a task when the victim actually has tasks left in its work queue. In contrast, Steal-back Mugging only mugs a victim when the thief attempts to reclaim previously stolen tasks. Consequently, the opportunities for consecutive successful steals may be more limited for Steal-back Mugging, which could explain its lower success rate.

Contrary to expectations, the Random Priority approach does not improve the number of successful steal attempts. In contrast, the Topology Priority approach provides a significant improvement. This difference may be explained by the cost of selecting a victim: by the time the Random Priority approach has identified the worker with the largest number of remaining tasks, those tasks may already have been executed by other thieves or by the victim itself. In contrast, the Topology Priority approach considers a smaller set of potential victims, allowing it to identify suitable victims more efficiently and reducing the likelihood that the selected victim has no tasks left in its work-queue.

Overall, the Random Mugging approach achieves the largest reduction in steal attempts and

the highest steal success rate for most benchmarks, followed closely by Topology Priority, which performs best on tsp and nqueens. The Register approach performs the worst of all approaches, combining a very high number of steal attempts with an extremely low success rate across every benchmark.

Approach		fib	adapint	tsp	nqueens	bh	matmul
Random	Total	59,832	77,379	53,030	49,544	601,563,477	19,406,817
	Successful	834	820	3,458	1,255	3,502	1,117
		(1.39%)	(1.06%)	(6.52%)	(2.53%)	(0.005%)	(0.0005%)
Topology	Total	5,580	8,330	6,418	3,752	26,616,974	1,034,745
	Successful	663	746	4,042	1,217	3,465	1,245
		(11.88%)	(8.95%)	(62.98%)	(32.44%)	(0.01%)	(0.12%)
Topology Priority	Total	34,746	20,613	36,595	15,723	22,182,909	835,550
	Successful	3,409	2,603	34,463	6,802	6,149	6,923
		(9.81%)	(12.63%)	(94.17%)	(43.26%)	(0.02%)	(0.83%)
Steal-back	Total	5,225,960	6,383,390	83,918	54,344	490,927,837	10,152,430
	Successful	1,278	1,379	3,977	1,711	3,935	1,633
		(0.02 %)	(0.02%)	(4.74%)	(3.15%)	(0.0008%)	(0.02%)
	steal-back	563	631	1601	735	1316	709
Steal-back Mugging	Total	132,795	136,492	245,576	126,331	194,710,880	1,636,504
	Successful	1,619	1,473	22,742	3,898	4,795	4,994
		(1.22%)	(1.08%)	(9.26%)	(3.09%)	(0.003%)	(0.04%)
	steal-back	433	361	1008	611	1076	673
Steal-back Hashing	Total	$2.55{\cdot}10^6$	$5.50{\cdot}10^9$	$1.37{\cdot}10^{10}$	$8.53{\cdot}10^8$	$1.89{\cdot}10^{10}$	$2.19{\cdot}10^9$
	Successful	947	52	31	53	46	53
		(0.04%)	($9{\cdot}10^{-7}$%)	($2{\cdot}10^{-7}$%)	($6{\cdot}10^{-6}$%)	($2{\cdot}10^{-7}$%)	($2{\cdot}10^{-6}$%)
	steal-back	418	21	0	22	15	22
Random Priority	Total	22,387	45,614	23,218	9,047	283,607,712	9,000,456
	Successful	227	235	1,413	1,255	419	1,633
		(1.01%)	(0.51%)	(6.09%)	(4.83%)	(0.0004%)	(0.005%)
Random Mugging	Total	796	972	30,332	3,601	4,165	4,374
	Successful	**793**	**959**	**30,128**	**3,569**	**4,124**	**4,343**
		(99.75%)	**(98.62%)**	**(99.33%)**	**(99.10%)**	**(99.02%)**	**(99.27%)**
Register	Total	44,189,395	24,156,984	822,433,238	93,647,724	237,600,953	65,978,251
	Successful	546	753	4,026	1,749	619	2,247
		(0.001%)	(0.003%)	(0.0005%)	(0.002%)	(0.0003%)	(0.003%)

Table 2: Total number of steal attempts and successful steal attempts per approach and benchmark.

Approach		*fib*	*adapint*	*tsp*	*nqueens*	*bh*	*matmul*
Random	Tot.	1.33	**1.44**	**1.35**	0.72	9575.13	762.14
	Per Task	0.05	0.06	0.01	0.02	90.48	16.56
Topology	Tot.	1.85	2.26	2.04	1.22	6585.17	**229.05**
	Per Task	0.08	0.11	0.02	0.03	67.41	7.41
Topology Priority	Tot.	8.79	10.70	32.53	5.01	**6368.45**	260.05
	Per Task	0.14	0.11	0.02	0.03	31.56	1.00
Steal-back	Tot.	6.90	17.32	1.74	1.36	10118.41	323.17
	Per Task	0.17	0.40	0.01	0.03	82.28	6.33
Steal-back Mugging	Tot.	5.49	8.15	8.36	5.11	9209.61	773.31
	Per Task	0.11	0.18	0.01	0.04	61.46	4.96
Steal-back Hashing	Tot.	4.75	9042.22	20435.72	1557.87	32456.97	3878.14
	Per Task	0.16	5564.44	21094.94	940.60	22578.76	2341.51
Random Priority	Tot.	**1.03**	2.22	2.12	**0.61**	10244.26	333.47
	Per Task	0.15	0.30	0.05	0.05	261.21	25.47
Random Mugging	Tot.	3.01	3.52	9.26	5.22	9022.30	712.70
	Per Task	0.12	0.12	0.01	0.05	70.0	5.25
Register	Tot.	5280.10	2440.64	96573.74	11171.45	30623.42	6045.24
	Per Task	309.68	103.61	767.52	204.39	1582.10	86.10

Table 3: The total number of milliseconds a thread is busy trying to successfully steal a task (Tot.), and the same value normalized per successfully stolen task (Per Task).

Table 3 presents the total time, in milliseconds, that workers spend attempting to successfully steal a task, as well as the average time required for a successful steal operation. The approach with lowest amount of time for each benchmark is highlighted in bold.

The table shows that, for the smaller benchmarks, Random Priority approaches are generally the fastest, while for the larger benchmarks, either Topology or Topology Priority is faster. Among the slowest approaches, Register and Steal-back Hashing each perform worst on different benchmarks, with no single approach being consistently the slowest. The poor performance of the Register and Steal-back Hashing approaches can be explained by inefficiencies in how these approaches select a victim, though the underlying cause differs between the two. For the Steal-back Hashing approach, computing the target costs a significant amount of time. For the Register approach, a likely explanation is contention on the lock protecting the queue of registered victims.

Figure 3 plots the steal-busy time against the number of steal attempts. These graphs show that for the smaller benchmarks (fib, nqueens, adapint, and tsp), the newly introduced approaches have a higher busy time than the random approach, but require fewer steal attempts. The exception to this is the Register and Steal-back Hashing approaches, which have both a very high busy time and a very high number of steal attempts. This changes when the benchmarks become larger (bh and matmul): the new approaches achieve either a lower busy time than the random approach, or a busy time comparable to it. A notable trend is seen in the Random Mugging approach: its

steal attempts remain relatively stable in comparison to the other approaches, but its busy time increases significantly on the larger benchmarks.

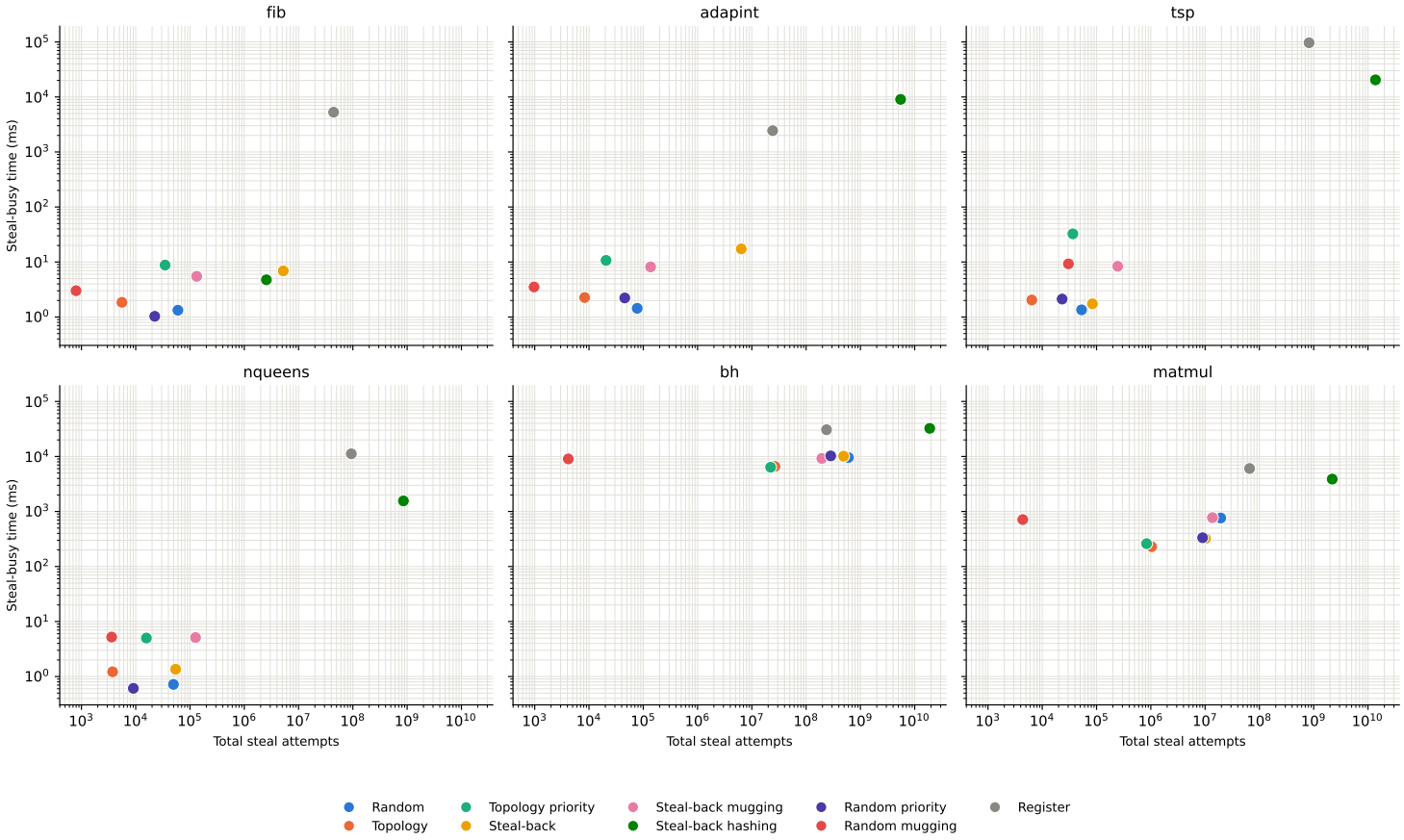

Figure 3: Relationship between number of steal attempts and time spent busy-stealing, broken down by scheduling approach and benchmark, measured at 32 threads.

5.4 Hardware performance

The cache performance is gathered using perf, every benchmark is run five times with 32 threads. The total number of cache references and cache misses is inspected in those five runs. Another hardware performance metric that is collected is the fraction of memory accesses that are remote. This fraction is calculated as: $fr\_remote = \frac{remote\_access}{(local\_access+remote\_access)}$, where local_access is the number of times the program fetches data from the local NUMA node and remote_access is the number of times the program fetches data from a remote NUMA node. This data is gathered for the random and topology-aware approach. A higher value of (fr_remote) indicates poorer memory locality, as a larger proportion of memory accesses are performed on remote NUMA nodes.

The data is gathered with perf using ls_refills_from_sys.ls_mabresp_lcl_dram:u for the number of local access and ls_refills_from_sys.ls_mabresp_rmt_dram:u for the number of remote access. Every benchmark test is run five times with 32 threads

Approach		fib	adapint	tsp	nqueens	bh	matmul
Random	References	852,019	$8.64 \cdot 10^6$	$1.46 \cdot 10^8$	$2.14 \cdot 10^7$	$1.04 \cdot 10^{10}$	$1.30 \cdot 10^9$
	Misses	187,278	207,865	640,067	206,543	$1.04 \cdot 10^9$	155,000,000
		(20.089)	(2.129)	(1.198)	(2.600)	(9.971)	(11.906)
Topology	References	$2.73 \cdot 10^6$	$9.25 \cdot 10^6$	$1.36 \cdot 10^7$	$6.86 \cdot 10^6$	$1.08 \cdot 10^{10}$	$1.39 \cdot 10^9$
	Misses	651,898	549,120	$1.35 \cdot 10^6$	650,192	$1.66 \cdot 10^9$	$2.15 \cdot 10^8$
		(22.989)	(7.714)	(2.953)	(5.646)	(15.404)	(15.895)
Topology Priority	References	$6.14 \cdot 10^6$	$9.16 \cdot 10^6$	$1.82 \cdot 10^7$	$5.23 \cdot 10^6$	$1.10 \cdot 10^{10}$	$1.38 \cdot 10^9$
	Misses	$3.26 \cdot 10^6$	$1.94 \cdot 10^6$	$4.10 \cdot 10^6$	$1.33 \cdot 10^6$	$1.74 \cdot 10^9$	$2.00 \cdot 10^8$
		(59.687)	(17.452)	(8.313)	(12.011)	(15.989)	(14.459)
Steal-back	References	$1.04 \cdot 10^6$	$5.59 \cdot 10^6$	$6.92 \cdot 10^6$	$1.34 \cdot 10^7$	$1.01 \cdot 10^{10}$	$1.30 \cdot 10^9$
	Misses	**257,335**	341,719	$1.67 \cdot 10^6$	272,202	$1.01 \cdot 10^9$	$1.43 \cdot 10^8$
		(19.741)	(5.089)	(1.794)	(2.557)	(9.986)	**(10.993)**
Steal-back Mugging	References	1,183,604	$8.36 \cdot 10^7$	$5.71 \cdot 10^7$	$6.32 \cdot 10^6$	$1.03 \cdot 10^{10}$	$1.30 \cdot 10^9$
	Misses	466,436	**611,937**	$1.94 \cdot 10^6$	933,056	$1.20 \cdot 10^9$	$1.48 \cdot 10^8$
		(33.890)	**(2.089)**	(3.971)	(10.554)	(11.445)	(11.383)
Steal-back Hashing	References	$1.88 \cdot 10^9$	$1.54 \cdot 10^8$	$5.27 \cdot 10^7$	$1.35 \cdot 10^7$	$9.67 \cdot 10^9$	$1.54 \cdot 10^9$
	Misses	$4.93 \cdot 10^8$	$1.35 \cdot 10^8$	**302,804**	$1.30 \cdot 10^6$	$6.29 \cdot 10^8$	$1.66 \cdot 10^8$
		(56.313)	(32.315)	**(0.043)**	(2.598)	**(5.617)**	(11.493)
Random Priority	References	781,571	$2.84 \cdot 10^6$	$6.86 \cdot 10^6$	$1.59 \cdot 10^7$	$1.02 \cdot 10^{10}$	$1.31 \cdot 10^9$
	Misses	201,583	183,877	$1.13 \cdot 10^6$	**253,300**	$1.09 \cdot 10^9$	$1.58 \cdot 10^8$
		(24.929)	(4.329)	(3.212)	**(2.542)**	(10.662)	(11.903)
Random Mugging	References	1,700,730	$5.19 \cdot 10^6$	$4.14 \cdot 10^7$	$7.54 \cdot 10^6$	$1.07 \cdot 10^{10}$	$1.33 \cdot 10^9$
	Misses	843,624	884,820	$1.09 \cdot 10^7$	$2.01 \cdot 10^6$	$1.41 \cdot 10^9$	$1.72 \cdot 10^8$
		(52.507)	(8.189)	(13.312)	(12.765)	(13.312)	(12.858)
Register	References	$6.72 \cdot 10^7$	$6.28 \cdot 10^7$	$2.99 \cdot 10^9$	$4.53 \cdot 10^8$	$1.02 \cdot 10^{10}$	$1.38 \cdot 10^9$
	Misses	$3.69 \cdot 10^7$	$2.81 \cdot 10^7$	$1.51 \cdot 10^9$	$2.45 \cdot 10^8$	$7.00 \cdot 10^8$	$1.84 \cdot 10^8$
		(35.085)	(20.890)	(53.319)	(74.778)	(6.906)	(12.836)

Table 4: Cache-references and cache-misses per approach and benchmark. This is the total amount across five runs.

Table 4 presents the number of cache references and cache misses for each benchmark and work-stealing approach. The approach with the lowest cache-miss rate for each benchmark is highlighted in bold.

The table shows that the random work-stealing approach already achieves a relatively low cache-miss rate, contrary to what might be expected from such a naive approach. The steal-back approach further reduces the cache-miss rate in benchmarks where the random approach exhibits relatively high cache-miss rates, such as fib and matmul. However, it does not lower that percentage by a large margin. Among the evaluated approaches, only the Random Priority approach consistently reduces the total number of cache references across all six benchmarks, but it does not consistently

decrease the miss rate.

Approach		fib	adapint	tsp	nqueens	bh	matmul
Random	Remote	967	854	76,392	2,600	16,291,665	3,676,844
	Local	1,443	1,084	66,750	1,980	46,373,621	4,504,583
	Fraction	**0,40**	0,44	**0,53**	0,57	0,26	**0,45**
Topology	Remote	3,048	1,675	285,384	2,226	14,123,519	4,168,943
	Local	2,022	2,591	169,066	2,452	54,382,454	4,837,241
	Fraction	0,60	**0,40**	0,63	0,48	**0,21**	0,46
Topology Priority	Remote	919	1,369	41,395	620	17,356,767	3,904,311
	Local	1,060	1,757	26,933	966	43,952,718	4,566,598
	Fraction	0,46	0,44	0,61	**0,39**	0,28	0,46

Table 5: Local and remote memory access for the random approach and topology approach.

Table 5 presents the number of remote and local memory accesses, as well as the remote access fraction (fr_remote) for the random and topology-aware approaches. The approach with the lowest fr_remote value is highlighted in bold.

Table 5 shows that the topology approach does not consistently improve NUMA locality. While it reduces the fraction of remote accesses for some bigger benchmark like the Barnes-Hut benchmark, it worsens or does not improve locality for several smaller benchmarks. The Topology Priority approach improves locality for some cases, but the benefits remain dependent on the benchmark characteristics. Overall, the results indicate that topology-aware victim selection can improve NUMA locality for workloads with significant memory access patterns, but its effectiveness depends strongly on the characteristics of the benchmark.

5.5 Performance

The parallel performance of the newly introduced approaches are evaluated in two ways. It is first evaluated, by calculating the speedup of the benchmark test (Fig. 4). Then to better show how much the speedup changes between the newly introduced approaches and the original random approach, the speedup relative to the random approach is also calculated (Fig. 5).

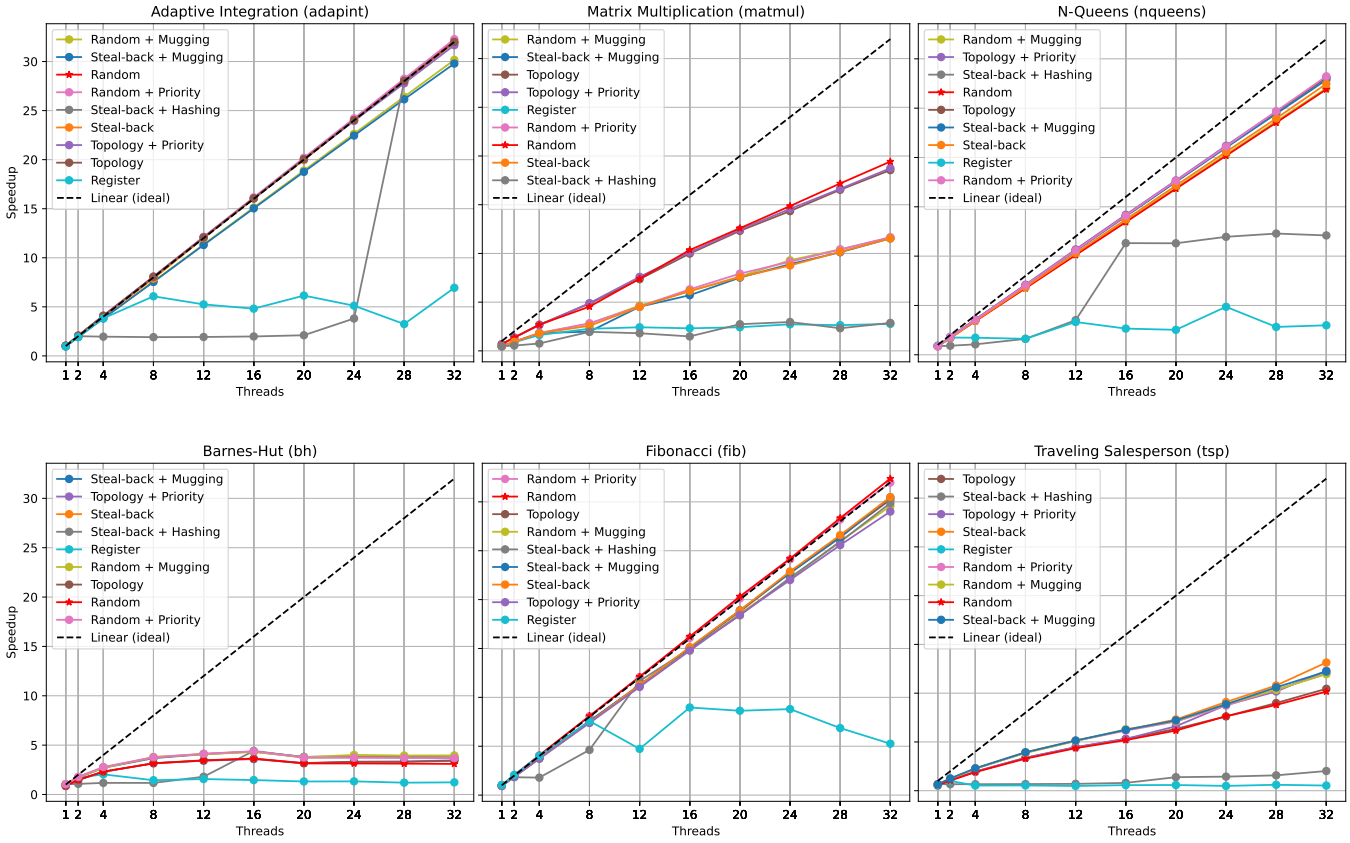

Figure 4: Parallel speedup of six benchmarks, parallelized with Velvet.

Fig. 4 shows that most approaches do not significantly change the speedup compared to the random approach. A notable exception is the poor performance of the Register and steal-back hashing approaches, which achieve substantially lower speedups than the other approaches.

Fig. 5 better illustrates that the newly introduced approaches do not significantly improve the speedup compared to the random approach. The two exceptions are the benchmarks with longer sequential run-times (reported in Table 1): Barnes-Hut and Traveling Salesperson, where the impact of the different approaches becomes more noticeable. But not one approach consistently improves the run-time across all the benchmark test. Another notable observation is the lower speedup achieved by most approaches for the Matrix Multiplication benchmark. Which is not reflected in previous done experiments.

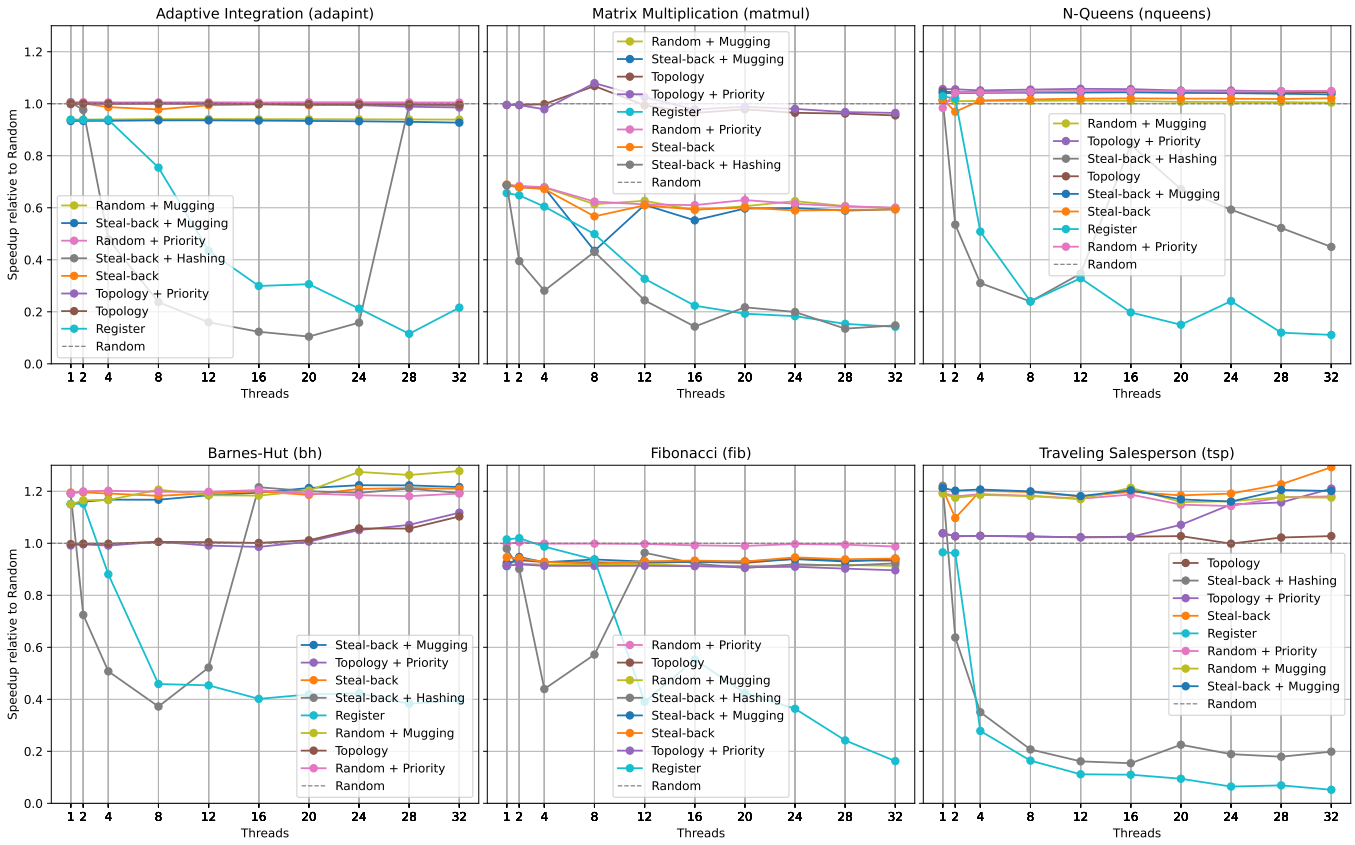

Figure 5: Parallel speedup relative to the random approach of six benchmarks, parallelized with Velvet.

6 Conclusion

In this thesis, several work-stealing approaches were investigated, each introducing an optimization over the baseline random work-stealing approach. The goal was to determine how these approaches affect the runtime of programs parallelized by Velvet. To understand why runtime is affected, multiple experiments were conducted. These experiments measured the number of steal attempts and successful steal attempts, as well as the average time a worker spends busy trying to successfully steal a task (Section 5.3), and the number of cache references and cache misses (Section 5.4). For the approaches that exploit hardware topology, the number of remote versus local memory accesses was also measured (Section 5.4).

Based on prior research, the steal-back approaches were expected to exhibit good data locality, reflected in a low percentage of cache misses. Similarly, the topology-aware approaches were expected to favor local memory accesses over remote ones. For the priority approach, a lower number of steal attempts and a higher fraction of successful steal attempts were expected.

These expectations were not fully met. The steal-back approaches improved data locality for some benchmarks (Table 4), but not consistently across all of them. Partly because the random approach already achieves good data locality in many cases. The Random Priority approach reduced the overall number of steal attempts. However, the fraction of successful steal attempts

did not significantly improve, and in some benchmarks it even decreased relative to the random approach. In contrast, the Topology Priority approach improved both the number of steal attempts and the fraction of successful steal attempts (Table 2). The approaches aimed at improving NUMA memory access locality succeeded in doing so, but again not consistently across all benchmarks (Table 5).

Although several approaches improved individual performance metrics, these improvements did not consistently translate into lower runtime. However, for benchmarks with longer sequential run-times, such as Barnes-Hut and Traveling Salesperson, the impact of the different approaches became more noticeable. The poor performance of the Register and Steal-back Hashing approaches was particularly evident, which is reflected in the average time a worker spends busy trying to successfully steal a task (Table 3).

The experiments showed that for benchmarks with a longer sequential runtime, the speedup of the new approaches was more visible than the speedup of the random approach. Because the supercomputer on which the experiments were conducted had a time limit of fifteen minutes, benchmarks with an even longer sequential runtime could not be tested. It would be interesting future work to test these approaches on benchmarks with even longer sequential run-times, to see whether the observed improvements continue to grow.

Another interesting direction for future work concerns the interaction between work-scheduling and work-stealing approaches. This thesis focused on work-stealing approaches, though work-scheduling approaches were also discussed. It would be worthwhile to examine how the two interact. For example, by combining work pushing (Section 3.1.1) with localized work-stealing (Section 3.2.3).

References

[1] Aaron Becker, Gengbin Zheng, and Laxmikant V. Kalé. *Load Balancing, Distributed Memory*, pages 1043–1051. Springer US, Boston, MA, 2011.

[2] Chenzhong Xu and Francis CM Lau. *Load balancing in parallel computers: theory and practice*, volume 381. Springer, 2007.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[4] B. Vikranth, Rajeev Wankar, and C. Raghavendra Rao. Topology aware task stealing for on-chip numa multi-core processors. *Procedia Computer Science*, 18:379–388, 2013. 2013 International Conference on Computational Science.

[5] Ryusuke Nakashima, Hiroshi Yoritaka, Masahiro Yasugi, Tasuku Hiraishi, and Seiji Umatani. Extending a work-stealing framework with priorities and weights. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 9–16, 2019.

[6] Seonmyeong Bak, Oscar Hernandez, Mark Gates, Piotr Luszczek, and Vivek Sarkar. Task-graph scheduling extensions for efficient synchronization and communication. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, page 88–101, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Carol Nichols Steve Klabnik and Chris Krycho. *The Rust Programming Language*. 2025.

[8] Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. *ACM Comput. Surv.*, 52(6), October 2019.

[9] Lukas Wirth. *The Little Book of Rust Macros*. 2025.

[10] B Vikranth, Rajeev Wankar, and C Raghavendra Rao. Affinity-aware synchronization in work stealing run-times for numa multi-core processors. In *Advanced Computing and Communication Technologies: Proceedings of the 11th ICACCT 2018*, pages 121–131. Springer, 2018.

[11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.

[12] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. An adaptive and hierarchical task scheduling scheme for multi-core clusters. *Parallel Computing*, 40(10):611–627, 2014.

[13] Derek L. Eager; Edward D. Lazowska; John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering (Volume: SE-12, Issue: 5)*, 1998.

[14] Badia Liokouras, Ben van Werkhoven, and Rob V. van Nieuwpoort. Velvet: Parallel divide-and-conquer in safe rust. In *European Conference on Parallel Processing (accepted for publication)*, 2026.

[15] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3), August 2014.

[16] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 125–137, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Blelloch G.E. Blumofe Acar, U.A. R.d. the data locality of work stealing. *theory of Computing Systems 35, 321–347 (2002)*, 2002.

[18] Michael Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, page 212–221, New York, NY, USA, 1998. Association for Computing Machinery.

[19] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3), August 2014.

[20] Saman Barghi and Martin Karsten. Work-stealing, locality-aware actor scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 484–494, 2018.

[21] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100–106, 2016.

[22] Brorsson M. Varisteas, G. Dvs: Deterministic victim selection to improveperformance in work-stealing scheduler. *MULTIPROG 2014: Programmability Issues for Heterogeneous Multicores*, 2014.