Universiteit Leiden

How does the performance and scalability of Velvet compare to the state-of-the-art?

Quinten Max Baris (s4084225)

Supervisors:
Dr. Rob van Nieuwpoort & PhD candidate Badia Liokouras

Bachelor Thesis– Informatica

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

July 22, 2026

Abstract

Parallelism is central to modern High Performance Computing, yet the traditional languages used in this field place memory management entirely in the programmer's hands, making correctness hard to guarantee. Rust offers comparable performance while enforcing memory safety at compile time, but its dominant parallelism library, Rayon, relies internally on large amounts of unsafe code. Velvet is a recently developed task-based, divide and conquer parallel programming library for Rust that achieves parallelism using only safe code below API level. This thesis evaluates how Velvet's performance, scalability and usability compare to the industry state of the art by porting three benchmarks to Velvet: the Embarrassingly Parallel (EP) and Integer Sort (IS) kernels from the NAS Parallel Benchmarks, and a Lattice Boltzmann fluid dynamics simulation (LBM) from SPEChpc 2021, chosen to represent a compute-bound, a memory-bound, and a mixed workload respectively. Each is compared against sequential and Rayon-based Rust and C implementations on up to 32 cores, with Velvet and Rayon each evaluated in variants with and without application-level unsafe code. The results show that Velvet is competitive with Rayon on all three workloads. On the compute-bound EP kernel, the fully safe Velvet implementation matches its unsafe counterpart at every thread count, demonstrating that fearless concurrency is attainable at no performance cost for this workload class. On the memory-bound IS kernel, Velvet with application-level unsafe code finishes within 1% of both Rayon and C at full core count. On the mixed LBM workload, the cost of full application-level safety only becomes visible once the computation spans both sockets of the test machine, at approximately 20% at 32 threads. The comparisons against C remain unsettled due to differences in single-thread code generation and compilation settings.

Contents

1 Introduction

Parallelism is a central topic in modern day High Performance Computing (HPC). For many decades, improvements in processor performance relied heavily on increasing clock speeds. However, this trend has stagnated due to physical limitations such as power consumption and heat dissipation [22]. As a result, modern processors no longer rely primarily on faster individual cores, but instead achieve higher performance by providing more cores and more opportunities for concurrent execution. Consequently, the main opportunity for performance improvement in modern HPC lies in the ability to parallelize workloads effectively.

For single node High Performance Computing, the OpenMP framework has been the industry standard choice. It is widely used with C, C++ and Fortran, especially because it lets the developer parallelize kernels using compiler directives without rewriting the whole program structure. This is especially desirable given the fact that C, C++ and Fortran have historically been the most widely used programming languages for High Performance Computing kernels.

However, these traditional languages place most of the memory management responsibility into the programmers hands. Given the fact that parallelized kernels heavily depend on perfect memory management, correctness of these kernels becomes hard to guarantee. Incorrect use of pointers, synchronization mechanisms, or shared memory access can lead to data races and other correctness issues [26].

The Rust programming language offers a modern alternative in this space. It is a systems programming language with comparable performance to C and C++ while also enforcing memory correctness and safety [16]. It enforces this with its signature borrowing system, enforced by a strict compiler. This prevents many common memory mismanagement errors and data races without relying on a garbage collector. This makes Rust an interesting candidate for High Performance Computing.

While the language provides strong safety guarantees, developers still need programming models and libraries that divide work over multiple cores and schedule this work efficiently. In the Rust ecosystem, Rayon is one of the most widely used libraries for shared-memory parallelism [22]. It provides a high-level programming model, allowing programmers to transform sequential iterators into parallel iterators with relatively small changes to the original program structure. Sometimes it can be just as easy as changing an iterator.

This raises the question of why a new parallel programming library is needed at all. The answer lies in how Rayon realizes its safe interface: internally, it relies extensively on unsafe code, the Rust keyword that bypasses the compiler's safety checks [18]. This has two consequences. First, this hidden unsafe code propagates into every program that depends on the library: even an application written entirely without the unsafe keyword inherits Rayon's unsafe internals through its dependency chain, and is therefore not guaranteed to be memory-safe by the compiler [18] and violates a crucial property [1] of Rust's promise of "fearless concurrency" [16, 1]. Second, the safety

[1] Fearless concurrency combines three properties: (1) concurrency errors are detected statically by the compiler, (2) the programmer does not need to resort to unsafe code, and (3) neither guarantee is paid for in performance [1].

of such a program no longer rests on the Rust compiler, but on the assumption that Rayon's unsafe code is correct. Since unsafe code can contain exactly the memory bugs that Rust was designed to prevent, and Rayon's implementation has not been formally verified in full [15], this assumption cannot be checked mechanically and must simply be trusted.

The Leiden Institute of Advanced Computer Science recently developed a recursive, divide-and-conquer task-based parallel programming library named Velvet [18], which addresses exactly this problem: it is implemented entirely in safe Rust, so its users inherit no hidden unsafe code. Rather than parallelizing data iterators, Velvet parallelizes recursive functions. It provides a task-based programming model, allowing programmers to parallelize recursive functions by annotating them as spawnable and letting a work-stealing runtime schedule the resulting tasks. In the simplest cases this requires little more than marking the function that is already being called recursively. The open question is what this full safety costs in performance, which leads to the research question of this thesis:

How does the performance and scalability of Velvet compare to the state-of-the-art?

To answer this question, this thesis ports three well-established HPC benchmarks to Velvet, representing a compute-bound, a memory-bound, and a mixed workload, and compares each against Rayon, sequential Rust, and the original C implementations on up to 32 cores. The results, presented in full in Section 8 and Section 9, show that Velvet is competitive with Rayon on all three workloads: on the compute-bound kernel the fully safe Velvet implementation matches its unsafe counterpart at every thread count, and the cost of full application-level safety on the mixed workload only becomes visible once the computation spans both sockets of the test machine.

1.1 Thesis overview

This bachelor thesis was conducted at the Leiden Institute of Advanced Computer Science (LIACS), under the supervision of Rob van Nieuwpoort and Badia Liokouras. The remainder of this thesis is structured as follows. Section 2 provides the necessary background on Rust, Rayon and Velvet. Section 3 discusses related work. Section 4 explains the three benchmarks, why they were chosen, and the building blocks of their implementations, after which Section 5 motivates the safe and unsafe variants evaluated and frames the fearless-concurrency perspective. Section 6.1 describes the Velvet-specific design choices, the experience of using the framework, and the performance hypotheses. Section 7 details the experimental setup and Section 8 presents the results. Finally, Section 9 answers the research question and Section 10 discusses directions for future work.

2 Background

2.1 Rust

Rust is a systems programming language that originated as a personal side project by Graydon Hoare in 2006 when he was still working at Mozilla. Then after 3 years in 2009 Mozilla started funding the development on Rust and finally it was presented to the public around 2010. The first stable version Rust 1.0 was released in 2015 [16].

Rust aims to solve the ever haunting problem of memory mismanagement. System languages such as C and C++ give the programmer full control over memory without checking if it is handled correctly. This inherently gives them the speed that they are known for. However, this also allows for bugs such as use-after-free, double-free, dangling pointers, data-races etc. Whom are hard to find because they often do not crash deterministically and besides that pose a big security threat. Roughly 60 to 70 percent of browser and kernel exploits can be traced back to memory mismanagement [26].

The answer to this problem has always been garbage collected languages such as Java, Go and C#. They eliminate most of these bugs by managing memory automatically at runtime. However, this can cause performance to deteriorate due to unpredictable pauses and garbage collection overhead. Rust solves this by enforcing strict rules at compile time to guarantee memory safety, eliminating all this overhead. It simply refuses to generate a binary if these rules are not met [16].

2.1.1 Ownership

Ownership is the backbone of Rust's safety model. The rules are short and simple. Every value has exactly one owner: when the owner goes out of scope, the value is dropped. Ownership can be terminally moved or temporally lent (borrowed). These rules replace garbage collection entirely [16].

Dropped when owner goes out of scope is conceptually copied from C++ with the RAII pattern (Resource Acquisition Is Initialization) [30, 12]. The compiler tracks ownership through scopes. When a variable goes out of scope, it generates a drop call of the specific value e.g. for a String the drop method frees heap memory etc. You do not call close() you just let the variable go out of

scope. You can trust this process because the compiler knows statically where each value scope ends so deallocation is deterministic and automatic. This way a double free or use after free for example cannot occur, because you yourself do not have to free the allocated memory.

2.1.2 Move Semantics

Rust lets you borrow a value through a reference without taking ownership, instead of passing it by value. A shared reference (&) allows aliasing by read-only shared access(as many as you want). And (&mut) allows a exclusive mutable reference. This is the borrow checking rule, a variable either has read only or read and write only access when aliased or referenced.

This constraint protects the developer from a lot of different mistakes: if no thread can hold a read reference while a writer is mutating, you cannot observe a half-updated value. If a container cannot be mutated while something inside it is borrowed, that inner reference cannot be invalidated underneath you. While you are borrowing something inside a vector, you cannot simultaneously mutate the vector, which could cause the borrowed element to move. Rust statically rejects code where a reference is used after a mutation, destruction, or move would invalidate it.

2.1.3 Lifetimes

Rust uses lifetimes to keep track of how long a reference to a value should stay alive. It is by definition the region of the program over which a reference is valid. It is then the borrow checkers job to verify that every reference's lifetime is fully in sync with the lifetime of the value it refers to [16].

Rust makes this analysis through non-lexical lifetimes(NLL) [11]: the borrow checker uses a combination of data flow analysis, and region inference to track the lifetimes of borrows and determine if they are valid. NLL works because it tracks where references are actually used, and ends the borrow after the final use instead of after the whole scope.

2.1.4 Unsafe Rust

Rust has one built in escape from it's strict compiler. An unsafe block permits a small set of operations that cannot be proven memory-safe by the compiler, such as dereferencing raw pointers or calling unsafe functions. All ordinary type, lifetime, and borrowing checks remain active. The programmer is responsible for ensuring that the additional safety invariants required by these operations are upheld.

2.2 Rayon

Rayon is the industry standard library for parallel programming in Rust [25]. Its central promise is: parallelism without pain. Rayon makes parallelizing sequential iterators easy. You change x.iter() in to x.par_iter(), and Rayon takes care of the rest "behind the scenes".

```rust
let x = vec![1.0, 2.0, 3.0];
let sum: f64 = x.iter()
    .map(|v| v * v)
    .sum();
```

Listing 1: Sequential

```rust
let x = vec![1.0, 2.0, 3.0];
let sum: f64 = x.par_iter()
    .map(|v| v * v)
    .sum();
```

Listing 2: Rayon

2.2.1 Functionalities

Rayon offers two ways of expressing parallelism. The first is data-parallel: a sequential iterator is replaced by one of Rayon's parallel iterators, after which operations such as map, filter and sum are distributed across the thread pool. Listing 2 shows that this often amounts to just one iterator change.

```rust
fn fib(n: u64) -> u64 {
    if n < THRESHOLD {
        return fib_fast(n);
    }
    let (r1, r2) = rayon::join(
        || fib(n-1),
        || fib(n-2));
    return r1 + r2;
}
```

Listing 3: Fork/join: Rayon join

The second is join, a fork/join primitive that takes two closures and executes them potentially in parallel. This is the natural fit for recursive divide-and-conquer, where the two branches of a recursive call are handed to join and the runtime decides whether to run them concurrently, as shown in Listing 3. The two mechanisms are not independent: Rayon's parallel iterators are themselves implemented on top of join, which recursively splits the underlying collection in half. join is therefore the more fundamental of the two.

2.2.2 Fundamentals

Beneath the visible API, Rayon's parallel iterators are built on the same join primitive. When a parallel iterator is consumed, the underlying collection is recursively split and each half is handed to a join call, so an iterator is ultimately executed as a recursive divide-and-conquer computation. The splitting is adaptive rather than fixed: Rayon continues to subdivide if and only if other threads are idle. If no is idle it continues to execute sequentially.

The semantics of join allow for very low overhead. When a worker calls join, it pushes the second closure onto its own double-ended queue and executes the first directly. If no other worker has taken the second closure by the time the first completes, it is simply popped again and run locally, so the cost of the join reduces to a push and a pop. Only when a task is actually stolen is any inter-thread communication incurred.

Load balancing is decentralised: an idle worker becomes a thief, selects a single victim uniformly at random, and attempts to take a task from the head of that victim's queue. This randomised strategy is not just a heuristic. Blumofe and Leiserson [8] prove that for fully strict computations, a randomised work-stealing scheduler executes a computation with work T_1 and critical path T_∞ in expected time $O(T_1/P + T_\infty)$, uses at most $S_1 P$ space, and incurs expected communication $O(T_\infty P S_{\max})$. Since both T_1/P and T_∞ are lower bounds for any P-processor execution, these bounds are within a constant factor of optimal, and the same guarantees are reported for the Cilk runtime [7].

These bounds are asymptotic and say nothing about the constants, nor do they guarantee good performance when there is not enough parallelism to exploit. Blumofe et al. [7] observe that once the number of processors approaches the average parallelism T_1/T_∞, the time spent attempting steals becomes a significant fraction of the total execution time. Fine-grained workloads with limited available parallelism are therefore the area in which work-stealing overhead becomes visible.

2.2.3 Unsafe code

Rayon promises is fully safe parallelism, no dataraces etc. But that safety is realized internally with a lot of unsafe code blocks [18]. However, a recognised end-to-end machine checked soundness proof does exist for the Rayon join primitive [15]. RustBelt includes a machine-checked proof for a model of rayon::join. This establishes soundness for the model, but does not constitute a verification of the complete current Rayon implementation, including its deque, latches and iterator infrastructure.

2.3 Velvet

Velvet is a parallel programming library built to parallelize sequential recursive functions [18]. What makes Velvet unique is that it is written entirely in Rust safe code and that it has a macro based interface. You simply annotate your recursive functions as spawnable and Velvet does the rest of the work for you.

2.3.1 Functionalities

```
fn fib(n: u64) -> u64 {
    if n < THRESHOLD {
        return fib_fast(n);
    }
    let r1 = fib(n-1);
    let r2 = fib(n-2);
    return r1 + r2;
}
```

Listing 4: Sequential

```
#[spawnable]
fn fib(n: u64) -> u64 {
    if n < THRESHOLD {
        return fib_fast(n);
    }
    let r1 = fib(n-1);
    let r2 = fib(n-2);
    return r1 + r2;
}
```

Listing 5: Velvet

Velvet has a macro based interface and an entire inherently safe Rust source code. It parallelizes recursive functions annotated with #[spawnable] 5 by generating new parallelized source code before compilation based on these annotations. Velvet relies on a build script to generate this code: in the build script one calls velvet::generate(paths) with every file containing spawnable functions. Velvet relies on the same divide and conquer and work stealing basis as Rayon for task scheduling and load balancing. The only differences are the queue back-ends. Velvets default setting is the "safe' queue, guaranteeing its no hidden unsafe code claims. However, it does contain two options to use unsafe queue back-ends: crossbeam and unsafe.

2.3.2 How is Velvet "safe"

Rayon needs unsafe because it's parallel iterators must split one datapiece (e.g. a &mut [T] slice) into disjoint chunks for every thread to mutate simultaneously. Implementing primitives that construct multiple disjoint mutable slices generally requires unsafe code internally, even though APIs such as split_at_mut expose the resulting operation safely. [18]. So it shares mutable state and must bypass the compiler to do so. Velvet's model is value-based rather than reference-based. A spawnable function takes Send + 'static arguments by value and returns a value by value. No shared mutable &mut is split across a thread boundary: instead, owned values are moved into the spawned task, and result values are returned at synchronization. This is exactly what Rust's ownership model permits without any escape hatch: one owner per value, transfer by move, and Send guaranteeing the transfer between threads is safe. This introduces other challenges when parallelizing a program which I will discuss in my conclusion 9. In conclusion: velvet chooses an API that stays within what the borrow checker can prove, whereas Rayon chooses an API that cannot.

2.3.3 Constraints

To be able to guarantee no hidden unsafe code and apply its macro based interface Velvet had to enforce some constraints in what datatypes could be used. Due to the fact that macros transform source code statically and to uphold thread safety all arguments to and from spawnable functions, in Rust terms must be Send + 'static [18]. A reference without Arc into the caller's stack frame satisfies neither: its lifetime ends when that frame does, while the task may outlive it.

Velvet therefore requires that by-reference arguments to spawnable functions are wrapped in Arc, Rust's atomically reference-counted shared pointer: if they are not, the program does not compile [18]. When the macro replaces a recursive call with a spawn, these arguments are cloned, which for an Arc increments the reference count, not perform a deep copy.

Recursive calls must also be uniform, explicit, and must not depend on locally declared variables. This is because information about local variables is not preserved when synchronizing spawned calls. For functions with return values, explicit returns eg. return var: are needed for velvet to be able to insert synchronization points correctly. And all toplevel calls of spawnable functions need to originate from the same velvet main function.

2.3.4 Fundamentals

Beneath the visible API, Velvet's spawnable annotation is a source-to-source transformation combined with a generated task representation. Before compilation begins, a Cargo build script scans the program for annotated functions, inspects their signatures, and emits an application-specific enumeration, the Velvet Frame, with one variant per spawnable function's arguments and one per return value. A procedural macro then rewrites the annotated function itself, replacing its recursive calls with spawn operations and inserting sync operations where data dependencies require them. The rewritten program is subsequently compiled by the unmodified Rust compiler and remains subject to its type, borrow and thread-safety checks [18].

As in Rayon, most spawned work is ultimately executed by the worker that created it, so the cost that matters is the cost of the non-stolen path. Velvet follows the work-first principle [14]: where the branch factor can be determined statically, the final recursive call is executed directly rather than enqueued, saving the construction of a VelvetFrame and the associated queue interaction. When a sync dequeues a task that was never stolen, the worker simply executes it by recursing, so no inter-thread communication is incurred.

Load balancing follows the same randomised work-stealing scheme described in Section 2.2: workers operate LIFO on the tail of their own queue while thieves take from the head, and the bounds of Blumofe and Leiserson [8] apply equally here. The mechanism by which a stolen result is returned, however, is shaped by the safety constraint. A thief replaces the stolen task in the victim's queue with a Stolen variant containing a locked VelvetFrame, executes the work while holding the lock, writes the result, and releases it. Releasing the lock acts as the completion signal. A worker that syncs on a task it finds still locked does not block, but begins stealing itself.

Where Velvet differs materially from Rayon is in the degree of concurrency the queue permits. Velvet's safe backend implements the work-queue protected in its entirety by a single lock, so owner and thief cannot access it simultaneously even though they operate at opposite ends. Rayon uses a lock-free Chase-Lev deque, which requires unsafe. This is a deliberate trade: compiler-verified thread safety in exchange for coarse-grained locking. Its measured cost is small, with a sequential threshold in place, work overhead is approximately 1.00 across the benchmark suite and the difference between queue backends is negligible, but it becomes visible when tasks are so fine-grained that queue interaction dominates useful work [18].

3 Related Work

3.1 NAS parallel benchmarks

The NAS Parallel Benchmarks (NPB) are a suite of programs developed by the NASA Advanced Supercomputing Division to evaluate the performance of parallel supercomputers [3]. The suite consists of both memory-bound and compute-bound kernels, each derived from real-world aerophysics workloads, designed to stress different subsystems of an HPC platform. Benchmarks are organised into problem classes (S, W, A through E) of increasing size, enabling measurements that range from single-node scaling tests to full-system evaluations. Each benchmark defines a fixed problem size and a nominal operation count per class, so that the reported Mop/s figure reflects only differences in execution time — not in the amount of work performed. This property makes NPB particularly well-suited for comparing programming models and runtime systems on identical hardware. Since

its introduction in 1991, the suite has become one of the most widely adopted benchmark standards in HPC research.

3.2 NAS Parallel Benchmarks in Rust

NPB-Rust by Martins et al. [22], ported the NAS Parallel Benchmarks [3] to Rust, using Rayon for parallelism. Their work continues an established methodology of porting NPB to evaluate programming models: the same research group previously produced NPB-CPP, a C++ version of the suite parallelized with OpenMP, TBB and FastFlow [19]. In their evaluation, sequential Rust was 1.23% slower than Fortran and 5.59% faster than C++, while Rust with Rayon was slower than both Fortran and C++ with OpenMP. They conclude that although Rayon is competitive in most cases and provides the benefit of memory safety, it was slower overall on the NPB suite, and explicitly identify the exploration of different parallelism methodologies in Rust as an open opportunity. This thesis addresses exactly that: it follows the same benchmarking methodology, but substitutes Rayon with Velvet. Moreover, the sequential Rust implementations of Martins et al. serve as the foundation on which the Velvet implementations of EP and IS in this thesis are built.

3.3 Fluid Dynamics simulation via Lattice Boltzmann method

The Fluid Dynamics simulation via Lattice Boltzmann method which this thesis ported from C to rust is originally developed by partners of the SPEC (The Standard Performance Evaluation Corporation), a non-profit corporation, which establishes and maintains standardized, vendor-agnostic benchmarks and tools to evaluate performance and energy efficiency of computing systems. SPEC was founded in 1988 by workstation vendors who had a need for realistic standardized performance tests. Today SPEC has more than 120 members among which are large hardware- and software vendors, universities, research facilities and government organisations. LBM is one of nine benchmarks in the SPEChpc 2021 suite [28]. Made by a research-team at the University of Ferrara and INFM(Istituto Nazionale di Fisica Nucleare). The benchmarksuite has been used by major vendors such as NVIDIA, Intel and Supermicro in product development for benchmarking.

3.4 D2Q37 performance studies on multicore architectures

The D2Q37 model underlying the SPEChpc LBM benchmark has itself been researched deeply by the Ferrara/INFN group that developed the code as partner of SPEC. Most relevant to this thesis is the work of Mantovani et al. [20], who use the D2Q37 LBM frame as a test case for identifying performance issues on multi- and many-core processors. Their analysis centers on the same two kernels described in Section 4.1.3: the memory-bound propagate step and the compute-bound collide step, and studies how data layout, vectorization and multi-threading affect the performance of each. Their central finding is that the two kernels place opposite demands on the hardware, due to this optimizing the memory access pattern for one may penalize the other.

3.5 Rust Rayon

Rayon itself was created by Niko Matsakis and Josh Stone [23] and, unlike the other frameworks discussed in this section, has no accompanying academic publication: it is developed as an open-

source community project and documented through its repository. Matsakis, at the time a member of the Rust core language team, designed Rayon explicitly around the Cilk-style work-stealing model discussed above, with the goal of demonstrating that data-race-free parallelism could be offered in Rust with minimal changes to sequential code. Since its initial release in 2015 it has grown into the industry standard for shared-memory parallelism in the Rust ecosystem [25], and it is used as the parallel baseline in most research evaluating parallel Rust, including Martins et al. [22], Abdi et al. [1] and this thesis.

3.6 The Velvet Framework

Velvet itself is developed by Liokouras et al. [18], a divide-and-conquer parallel programming model implemented entirely in safe Rust, which adopts a Cilk-style spawn–sync abstraction with work-stealing scheduling and integrates with the standard Rust toolchain. The framework is motivated by the fact that Rayon, the industry standard for shared-memory parallelism in Rust, relies extensively on unsafe Rust code in the API backend. Velvet uses zero unsafe code, when opting for the safe queue backend option, and thus guarantees 100% memory safety when using its default backend queue. However, Velvets goals is not 100% safe Rust applications but no unknowingly inherited unsafe Rust for its users. Their experiments showed that Velvet incurs minimal overhead despite its fully safe design and achieves competitive performance and scalability with Rayon, Cilk and OpenMP on up to 128 cores. However, this evaluation was conducted on six divide-and-conquer algorithms: Fibonacci, adaptive integration, Traveling Salesperson, Barnes-Hut, matrix multiplication and merge sort. These are valuable microbenchmarks, but they are not a standardized scientific benchmark suite. The authors explicitly note that applying Velvet to large-scale, real-world applications is a natural next step to validate its practicality. This thesis takes up this direction by evaluating Velvet on 2 benchmarks of the NPB benchmark suite, a recognized standard for HPC benchmarking. And a Lattice Boltzmann fluid dynamic simulation developed by a well respected performance engineering organisation SPEC (The Standard Performance Evaluation Corporation).

3.7 Cilk and task-based parallel programming

The spawn-sync programming model that Velvet adopts is inspired by Cilk [18], a multithreaded extension of C developed at MIT [7]. In Cilk, the programmer exposes parallelism by marking function calls as spawnable and inserting sync points where their results are needed, while a randomised work-stealing scheduler distributes the resulting tasks over the available processors. The theoretical foundation of this model was provided by Blumofe and Leiserson [8], who proved that work-stealing executes fully strict computations within a constant factor of optimal time and space, and its practical efficiency by Frigo et al. [14], whose work-first principle reduced the cost of a spawn to a small constant multiple of a function call. This combination of a minimal programming interface with provably efficient scheduling made the Cilk model highly influential: it underpins later industrial task-based frameworks such as Intel Threading Building Blocks [27] and the tasking model introduced in OpenMP 3.0 [2], and, in the Rust ecosystem, both Rayon's join primitive and Velvet itself. Velvet can therefore be seen as bringing the Cilk model to Rust in its most direct form, with the distinguishing property that its implementation stays entirely within safe Rust.

3.8 Rust in HPC

Moran and Bull [24] evaluates the suitability of Rust for HPC systems by implementing a simple computational fluid dynamics code in Rust, C and Fortran. Comparing sequential and parallel versions across multiple problem sizes. For the sequential problems they conclude that Rust is as performant as C and Fortran. For the parallel versions, however, the Rust implementation was slower: the strict memory-safety rules did not permit the sharing of mutable state between threads, forcing the implementation to duplicate arrays to guarantee the absence of data races. This finding prefigures a central problem / theme of this thesis. The cost they observe is not caused by the parallel runtime, but by the synchronization and copying that safe Rust forces on the application whenever a workload does not decompose into independently owned pieces of data. This is exactly the cost that Rayon hides inside its internally unsafe implementation and that the C baselines are not subject to at all, and that a fully safe framework such as Velvet must confront directly.

3.9 High-level structured stream parallelism in Rust

Pieper et al. [25] explore an alternative direction for parallelism in Rust by introducing Rust-SSP, a high-level abstraction for structured stream parallelism on shared-memory multicores. Instead of parallelizing data iterators or recursive functions, Rust-SSP expresses a computation as a pipeline of stages through which a stream of items flow, following the structured parallel programming approach that is well established in C++ but largely unexplored in Rust. Alongside the abstraction, the authors made a benchmark suite of stream processing applications and compare Rust-SSP against Rayon, Tokio and manual implementations with standard library threads. This work is relevant to this thesis for two reasons. First, it demonstrates that Rayon's iterator-centric model is not the only viable parallel programming abstraction for Rust, and that models targeting a specific class of workloads can outperform a general-purpose library within that class. Second, it shares Velvet's design philosophy of a high-level interface: the programmer declares the parallel structure and the runtime handles scheduling. Where Rust-SSP targets streaming pipelines, Velvet targets recursive divide-and-conquer, so the two approaches cover complementary regions of the workload spectrum that Rayon serves with a single generic model.

3.10 Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body

Costanzo et al. [10] compare Rust and C on the gravitational N-Body problem, a compute-bound kernel popular in the HPC community. Starting from a sequential Rust implementation as a base, they parallelize the hot kernels with Rayon parallel iterators. They measure each in isolation before comparing the result against a tuned OpenMP C baseline. In double precision[2] both versions perform practically the same, while in single precision C is up to 1.18× faster, which the authors claim is due to the compiler translating relaxed floating-point operations more effectively rather than to the language itself. Interestingly, block processing gave no improvement in Rust, whereas the C version needed explicit restructuring to exploit data locality the authors credit this to Rust

[2]Single and double precision refer to the IEEE 754 binary32 and binary64 floating-point formats, corresponding to `f32`/`float` and `f64`/`double`. Halving the width doubles the number of elements that fit in a cache line or SIMD register, at the cost of roughly seven significant decimal digits instead of sixteen.

iterators handling locality implicitly. Alongside performance they also evaluate programming effort by lines of code: 40 lines for the Rust main block against 66 for C.

3.11 When Is Parallelism Fearless and Zero-Cost with Rust?

Abdi et al. [1] question whether Rust's promise of "fearless concurrency" actually holds across all types of parallelism. They port 14 benchmarks from the Problem Based Benchmark Suite (PBBS) from C++ to Rust, using Rayon, and classify each parallel access pattern by how much "fear" it leaves the programmer with: fearless when concurrency errors are caught at compile time, comfortable when they surface at run time close to their cause, and scared when they occur undetected. Their conclusion is that Rust provides fearlessness only for regular parallelism, where data dependencies are statically known. For irregular access patterns, where write locations depend on run-time values, the programmer must choose between unsafe code, unnecessary synchronisation, or dynamic checks with substantial overhead. Across their suite, only 11% of accesses are supported by safe Rust alone, 60% rely on interior-unsafe library functions with static checks, and the remaining 29% are either unsupported or require run-time checking.

Two of their findings are directly relevant to this thesis. First, they confirm that Rayon's fearlessness rests on interior-unsafe functions, this is precisely the property that motivates Velvet's fully safe design [18]. Second, although their Rust benchmarks were 1.09× faster than the C++ versions at a single thread, they were 1.44× slower at 24 threads and scaled worse on every benchmark. The authors attribute part of this gap to Rayon's runtime management being less effective than Cilk's.

4 Approach

To answer this thesis's research question: How does the performance and scalability of Velvet compare to the state-of-the-art? This thesis will be evaluating Velvet's performance on three well respected benchmarks, two from the well known NPB benchmark suite originally developed by NASA and later ported to Rust by M.Martins et al [22], and one from the SPEChpc 2021 benchmark suite [28]. This thesis will be comparing an parallel implementation of these benchmarks with Velvet against their original C sequential and parallel implementation, a sequential Rust implementation, and a parallel Rust implementation with Rayon.

4.1 Benchmark Explanation

4.1.1 Embarrassingly Parallel: NPB

EP is particularly suitable for parallel execution. It almost entirely compute-bound so it is executed on multiple threads with minor cross thread communication and synchronization, making it an almost ideal parallel workload.

The EP benchmark generates an extremely large number of Gaussian-distributed values. It first produces pairs of uniform pseudorandom numbers, rescaled to the range $(-1, 1)$ so that each pair forms a point in the unit square. The kernel then checks whether each point falls inside the unit

circle. This acceptance rejection step is mathematically necessary rather than a mere sampling trick: the subsequent transformation takes the square root of $-2\ln t/t$, where t is the squared distance from the origin, and this expression is only well-defined for $t \leq 1$. For every accepted point, this transformation,known as the Marsaglia polar method [21], converts the two uniform coordinates into two independent Gaussian-distributed values. The benchmark adds these values to running sums and tallies each pair into one of ten histogram bins according to $\max(|X_k|, |Y_k|)$, so that each bin corresponds to a square annulus around the origin [3].

The kernel generates 2^M pairs of uniform pseudorandom numbers, or 2^{M+1} individual random numbers. Class C uses 2^{32} pairs, corresponding to approximately 4.29 billion pairs and 8.59 billion individual random numbers. Class D uses 2^{36} pairs, corresponding to approximately 68.72 billion pairs and 137.44 billion individual random numbers. Since a pair is accepted when it falls inside the unit circle, the expected acceptance fraction is $\pi/4$, yielding approximately 53.97 billion accepted pairs for class D.

To execute this in parallel with minor cross thread communication the total work is divided in to batches where each batch processes 2^{16} random number pairs. Each batch is assigned to a single thread, and that thread handles the entire computation for that batch independently. No other thread is involved or has communication at any point during this process. This is possible due to each batch having its own unique starting position in the global random number sequence. Each thread can compute this index from just its batch index.

The only moment threads communicate is at the end when all local sums and histogram counts are added together into a single final result. This reduction step is minimal compared to the actual computation so it barely affects overall performance.

4.1.2 Integer Sort: NPB

Unlike EP, IS is a communication-heavy and memory-bound kernel rather than a compute-bound one. Where EP demonstrates how a workload behaves when threads barely interact, IS stresses the opposite end of the spectrum: it depends heavily on how data is partitioned across threads and how efficiently those threads can read from and write into shared memory. It is therefore a good measure of how well a parallel runtime handles irregular memory access patterns and the synchronization that comes with them.

In the IS benchmark, a large array of integers is sorted. The keys are made by summing four consecutive numbers from a large list of randomly generated numbers and then rescaling the result. Resulting in a approximately Gaussian distribution between zero and a maximum key value. The maximum key value and total number of keys depend on the problem class. For this thesis class C and D are chosen.

IS sorts these keys not by comparing them but by using the Bucketsort algorithm, which uses the pre-knowledge that the keys are integers within a known range. It works by computing a key's final rank by counting how many keys are smaller than or equal to it, without ever directly comparing two keys. The benchmark does this in several stages. First, the keys are divided into a fixed number of buckets, and each key is classified into a bucket by taking its most significant bits (the upper bits of the key, obtained by shifting right). Once the buckets are known, the keys are

physically moved so that all keys belonging to the same bucket sit contiguously in memory. Finally, within each bucket the keys themselves are used as indices into a counting array: each occurrence of a key increments the count at that key's position, and a running cumulative sum over those counts turns the raw populations into ranks. The final output of the kernel is this rank array, and selected entries of it are checked against known correct values to verify correctness. The kernel repeats this ranking ten times, with two keys modified before each iteration so that the verification values shift in a predictable way. This so that there exists no implementation that accidentally produces the right answer for a fixed input.

To execute this in parallel the work is split along two different dimensions depending on the stage. During the counting and scattering stages the key array is divided into equal chunks, with one chunk assigned to each thread, so that every thread counts and places only the keys in its own region. During the final ranking stage the work is instead divided by bucket, so that each thread takes ownership of a disjoint set of buckets and ranks all keys within them. Because the buckets are independent of one another, this final stage is itself embarrassingly parallel once the keys have been distributed. However, threads must communicate at the boundaries between stages. Each thread first counts how many keys fall into each bucket within its own chunk, producing a per-thread histogram of bucket sizes. These per-thread histograms must then be combined so that every thread knows the global offset at which its keys should be written: a thread computes, for each bucket, how many keys earlier threads contributed to lower buckets and to the same bucket, and only then can it scatter its keys into the correct global positions without colliding with another thread's output. This offset computation is the essential synchronization point of the benchmark because it determines a unique destination index in shared memory for every single key. After the keys are scattered, the per-bucket ranking can again proceed independently.

4.1.3 Lattice Boltzmann Fluid Dynamics simulation: Spec2021 HPC

Computational Fluid Dynamics (CFD) is the numerical simulation of the flow of liquids and gases. Because the underlying equations of fluid motion generally admit no analytical solution. Practical questions, such as the airflow around an aircraft wing, heat convection in a reactor, or weather and climate dynamics can only be answered by discretising space and time and computing the flow's evolution numerically. These simulations are one of the most demanding workloads in high-performance computing: achieving physically meaningful resolutions requires updating billions of grid points over many thousands of time steps, which is why CFD codes have historically driven the development of supercomputers and remain a core component of HPC benchmark suites, including the original NPB [3]. The Lattice Boltzmann Method (LBM) is a class of CFD algorithms that, rather than discretising the macroscopic flow equations directly, models a fluid as populations of particles that repeatedly propagate between the sites of a regular lattice and collide at each site. This formulation makes LBM interesting as a benchmark: each time step consists of a memory-bound propagation phase and a compute-bound collision phase over a regular grid, stressing both the memory subsystem and the floating-point units of a machine, while the site-local collision step exposes abundant parallelism.

Lattice Boltzmann schemes are generally memory-bound. In common models such as D2Q9 or

D3Q19[3], each population is read and written exactly once per time step and is not reused within an iteration, so the ratio of arithmetic to transferred data is low.

The D2Q37 model used in this thesis differs from this pattern. Its 37 populations per site and its collision operator, which reproduces the full thermo-hydrodynamical behaviour of a fluid obeying the equation of state of a perfect gas [28], raise the arithmetic per lattice site a lot: the collision kernel performs 6606 floating-point operations per site. The benchmark therefore comprises two kernels with opposite characteristics: propagate is strongly memory-bound, whereas collide is compute-bound with an arithmetic intensity of approximately 11 FLOP/byte [28]. This combination is precisely what makes the benchmark suitable for evaluating a parallel runtime: a single application exercises both the memory subsystem and the floating-point units of the target machine.

The simulation operates on a two-dimensional grid of cells. Each cell stores a set of 37 numbers, called populations, one for each of the discrete directions of the D2Q37 model(two dimensions 37 populations). Together these populations describe how much fluid mass is moving in each direction at that cell. The entire state of the simulation is therefore the grid of cells, each holding its 37 populations, the simulation advances this state forward one time step at a time, repeated for a fixed number of iterations.

The LBM simulation has 2 phases: collision and propagation.

The collision step is purely local: it updates the 37 populations of a cell using only data already present at that cell, without consulting any neighbour. Where propagation moves populations between cells, collision determines how the populations at a single cell relax towards their local equilibrium. It first reduces the 37 populations into a compact description of the local fluid state, then computes the equilibrium populations corresponding to that state, and finally pushes the current populations a controlled amount towards that equilibrium. Because this depends on nothing outside the cell, every cell can be collided completely independently.

The simulation performs this collision through three separate invocations of the collide kernel, distinguished by which region of the grid they cover and how streaming is handled. The first handles the bulk of the domain: the interior cells that lie far enough from both walls that all of their neighbours are valid fluid cells. For these cells the streaming and the collision can be combined into a single pass, since every population a cell needs can be gathered directly from its surrounding neighbours.

The cells within a few rows of the top and bottom walls cannot be treated this way. They would go out of the physical domain, into regions that hold no valid fluid data, so the populations streaming in from outside the wall must first be supplied by a boundary condition derived from the prescribed state of the wall. For these wall strips the simulation therefore separates the steps: streaming is performed first by a dedicated propagation pass, the boundary condition then fills

[3]In the DnQm notation, n denotes the number of spatial dimensions and m the number of discrete velocity directions, with one population stored per direction at every lattice site.

in the populations entering from outside the domain, and only afterwards the collision is applied. Because the streaming has already been done at this point, this final collision operates in place on the cell's own populations instead of gathering from their neighbours.

Collision is the computationally heavy part of the simulation. It performs a large number of floating-point operations per cell across its several passes over the population array, yet because it reads and writes only local cell data, it requires no communication between threads and can be parallelized over cells without any cross-thread dependency. This is precisely the property that gives LBM its compute-bound, embarrassingly parallel aspect, in direct contrast to the propagation step.

The propagation/streaming step is the step where data gets moved between cells. After the populations of a cell have been updated locally, each of the 37 populations is sent to a neighbouring cell, in the direction associated with that population. A population going east moves to the cell to the east, a population going to a diagonal neighbour two cells away moves there, and so on, up to a maximum of three cells that the D2Q37 model allows. This results in every cell being updated with its "received" populations.

Propagation is memory-bound because it transfers a large amount of data while performing little floating-point computation.

These steps combined make one iteration. This is then repeated for the set number of iterations. After all iterations have completed, the combined mass of all cells before the simulation is compared to the combined mass after the simulation. If this relative error is smaller then 10^{-10} then verification is passed and the simulation is completed successfully.

4.2 Benchmark choice motivation

To evaluate Velvet's performance against the state-of-the-art, this thesis uses the above explained benchmarks. Since Velvet is a task-based library, its performance is expected to strongly rely on how well a certain workload can be expressed as independent tasks. For that reason, the benchmark selection is not only based on whether the programs are well-known HPC benchmarks, but also on whether they stress different aspects of parallel execution: independent computation, shared-memory communication, and a combination of both.

The first benchmark: Embarrassingly Parallel, represents the most ideal case for a task based parallel workload due to it being almost entirely compute-bound. This makes it useful as an upper-bound scalability test: if a parallel runtime performs poorly on EP, the cause is likely to be scheduling overhead, task granularity, or reduction overhead rather than memory sharing or synchronization. EP is therefore chosen to evaluate whether Velvet can exploit highly independent parallelism efficiently.

The second benchmark: Integer Sort, represents a much less favorable case for Velvet. Unlike EP, IS is dominated by memory access and thread synchronization / communication. The benchmark requires threads to coordinate through shared data structures and global offsets, which makes

it a good stress test for workloads where parallel performance depends on memory bandwidth and communication rather than pure computation. This is especially relevant for Velvet, because Velvet's safe value-based model does not allow the same style of shared mutable access that Rayon can express more directly. IS therefore tests whether Velvet can still perform well when the workload does not naturally decompose into independent owned tasks.

The third benchmark: the Lattice Boltzmann method fluid dynamics simulation alternates between a local collision step, which is compute-heavy and can be parallelized over cells, and a propagation or streaming step, which moves population values between neighboring cells and stresses memory bandwidth. The SPEC description of 605.lb_s explicitly identifies these two kernels: the propagate kernel is strongly memory-bound, while the collide kernel is compute-bound with an arithmetic intensity of approximately 11 FLOP/byte [28]. This makes LBM particularly useful because it does not only test an artificial extreme, but combines both computational and memory-bound behavior within one realistic scientific simulation.

The combination of these benchmarks with different workload types make it possible to not only evaluate whether Velvet can be fast, but also which types of workloads it can be competitive with the state-of-the-art and where its restrictions become a performance bottleneck.

4.3 Implementation

In this section we only discuss the building blocks, not how the benchmark where ported. Programming methodology's are discussed in the section 6.1

4.3.1 Embarrassingly Parallel and Integer Sort: NPB

These NPB benchmark had already been ported to rust by M.Martins et al [22]. The Velvet implementations are built further up on their sequential Rust implementations. The body's of the recursive Velvet functions are inspired by the Rayon iterators of their parallel implementations.

4.3.2 Lattice Boltzmann Fluid Dynamics simulation: Spec2021 HPC

The LBM C implementation had different header files for every step of the simulation as explained before in 4.1.3 and for the main loop and constants declarations. For the Rust implementation the same structure was adopted to keep oversight. First the sequential version was implemented just simply copying the C source code but with Rust syntax. The parallel Velvet implementation was built op on this sequential foundation.

The Rayon implementation was also built upon the sequential foundation. In each compute step the result structure of arrays was split into mutable chunks with an unsafe pointer slice from the global result. The only other change was the outer for loops of each kernel to into_par_iter() closures.

To count the number of FLOPS global constant values FLOPS_PER_SITE and FLOPS_PER_BOUNDARY_C (details of these FLOP counts can be found in source code). These are the counted number of Floating point operations done per section and combined are the total number of FLOPS done per

iteration. Because the compiler might reschedule some operations outside the fluid dynamics loop between the pure fluid dynamics simulation. Dividing the total number of operations by runtime is a more saturated metrics do use for comparing frameworks.

5 Methodology

To answer the thesis's research question: How does the performance and scalability of Velvet compare to the state-of-the-art? I extended the research further then originally planned in the research proposal. The original planned experiments where: comparing Velvet's performance on NPB EP and IS to their C and Rayon implementation. And to report on just the experience/usability of using Velvet for a parallel implementation of Lattice Boltzmann Fluid Dynamics simulation. This because the original SPEC LBM implementation is large (+/- 2500 LOC excluding MPI code) and complex. It was also believed to be exclusively memory-bound due to the general LBM models such as D2Q9 and D3Q19 being so. And since IS is already memory-bound, also comparing to Rayon here was deemed unnecessary.

However, this assumption proved to be incorrect. After obtaining the C source code from SPEC it became evident that the D2Q37 model used in SPEChpc 2021 does not follow this pattern: its 37 populations per site and its collision operator raise the arithmetic intensity to the point where the collide kernel is compute-bound, while only propagate remains memory-bound [28]. LBM therefore exercises both cases within a single application. This is a mix that was not yet covered in this thesis so I wanted to also evaluate performance here against Rayon and C.

Reading Abdi et al. [1] motivated a second perspective on this work. The concept of *fearless concurrency* [16] combines three properties: concurrency errors are detected statically by the compiler, the programmer does not need to resort to unsafe code, and both guarantees are not paid for in performance. Abdi et al. show that these properties only hold all together for regular parallelism. Whenever tasks must mutate aliased state, Rust's alias mutability forces a choice between unsafe code and dynamic checks whose overhead, and errors appear at run time [1]. Rayon violates the second property at the library level: its safe interface is realized internally through unsafe code [18], so the guarantee rests on the correctness of the implementation instead of on the compiler alone.

Velvet is interesting in this perspective because in its source code it introduces no unsafe code whatsoever [18], and the programmer therefore inherits none when using Velvet. The open question is the third property. If a Velvet implementation has performance competitive with C and with Rust using Rayon, this would indicate that the safety guarantee does not trade against performance, and thus that fearless concurrency is attainable for this class of workloads rather than just claimed. Consequently, a performance gap would locate the cost of full safety, which is an equally informative result for Velvet.

Besides Velvet's purely "safe" usage, this thesis also evaluates the performance of Velvet using unsafe code. Velvet's goal is not purely safe Rust on itself. Its goals is to provide a parallel programming framework with whom the programmer does not inherit unsafe code [18]. Besides

this, comparing a fully safe implementation against internally unsafe alternatives conflates two distinct costs: the overhead of the parallel runtime and the overhead of the synchronization that safe Rust forces upon the application. A Rayon program appears free of the latter only because the framework's internal unsafe code absorbs it, and the C reference is not subject to it at all. Thus it is necessary to evaluate both safe and unsafe implementations with Velvet to ensure fairness and to ensure the third "fearless" property [1].

For the same reason, the Rayon implementations are also evaluated in a safe and an unsafe variant. Restricting the comparison to safe Velvet against Rayon with unsafe application code would be unfair and unaligned with the two costs the previous paragraph distinguishes, since the safety condition would then differ on both sides of the comparison. Evaluating both frameworks under both conditions instead makes a comparison in which only one variable changes at a time: at the application level, the cost of the synchronisation that safe Rust imposes, and between frameworks, the cost of the parallel runtime itself.

It should be noted that a safe Rayon application is not equivalent to a safe Velvet application. Rayon's parallel iterators expose an interface that appears safe while relying on unsafe internals [18], so a Rayon program without application-level unsafe code still inherits unsafe code through its dependency, which is precisely the propagation that motivates Velvet's design. The safe Rayon variants therefore measure what a programmer can achieve without writing unsafe code themselves, not what can be achieved without unsafe code being present.

Together, these perspectives of evaluation require Velvet, Rayon and C versions of each benchmark: the fearless concurrency perspective requires a fully safe Velvet version and an unsafe Velvet version. However, given that Martins et al. [22] did not supply a fully safe parallel IS implementation[4]:. The `--cfg safe="true"` flag only flips the `UNSAFE` constant at `is.rs:114--117`, which gates three `if UNSAFE` branches choosing between `get_unchecked` and normal indexing (`is.rs:508`, `553`, `601`). The structural `unsafe` code, however, runs unconditionally in both modes:

- `unsafe impl Sync for UnsPtr` at `is.rs:112`;
- five `std::slice::from_raw_parts_mut` calls inside `par_iter` closures — `is.rs:326` (`create_seq`), `417`/`419` (in-bucket sort), `523` and `579` (`rank`) — each of which hands every Rayon thread its own full `&mut` view of the same shared buffer.

After discussion with my second supervisor I decided not to make a IS Velvet fully safe implementation. Therefore this thesis cannot conclude on the fearless concurrency perspective of Velvet on a fully memory-bound kernel [3]. This is something added to future work 9.

Thus we conduct experiments with the following implementations:

- **EP**: Rayon safe and unsafe, Velvet safe and unsafe and C.
- **IS**: Rayon unsafe, Velvet unsafe and C
- **LBM**: Rayon safe and unsafe, Velvet safe and unsafe and C.

[4]Line numbers refer to the NPB-RAYON/is.rs in the NPB-Rust repository, https://github.com/GMAP/NPB-Rust

Also, all Velvet implementations in this thesis are run with the default Safe queue backend. Liokouras et al. [18] evaluated all three supported backends and found that the differences are negligible for realistic workloads and only become visible under extremely fine-grained tasks where framework overhead dominates. Since the benchmarks in this thesis operate at far more coarse grained task granularity than the benchmarks exposing this effect, this thesis does not evaluate the other queue backends. Also, this reflects the configuration under which Velvet's central claim: no inherited unsafe code [18] actually holds.

Besides pure performance evaluation we will compare the difference in lines of code of each implementation and report on our own experience using velvet to evaluate programming effort differences between Rayon and Velvet.

6 Velvet design choices, experience with Velvet, framework suggestions and Hypothesis

6.1 Velvet design choices

To stay within Velvet's 'static and Send bounds, all arguments to spawnable functions must be owned: &mut references cannot cross a spawn boundary [18]. Shared data structures were therefore wrapped in Arc, Rust's atomically reference-counted shared pointer. When Velvet replaces a recursive call with a spawn, these arguments are cloned, which increments the reference count instead of performing a deep copy, so the underlying data is shared instead of duplicated per task. Unlike the borrow checker's static analysis, this reference count is a runtime mechanism: the compiler cannot determine at compile time which task will finish last, so the value is dropped when the count reaches zero rather than at the end of a lexical scope.

An Arc alone, however, only grants shared read access. To also allow the spawned tasks to write, interior mutability is required. The data structures that are both read from and written to therefore need to be stored as Arc¡Vec¡AtomicU64¿¿, where the atomic elements provide the mutability that Arc itself does not. However, NPB EP and LBM use f64, this is not a problem because they can be easily casted to and from u64 with Rust's f64::to_bits() and f64::from_bits() functions.

Given EP and the collide kernel in LBM their embarrassingly parallel nature the Velvet with safe application level code did not have to make use of compare and exchange or fetch_add atomics which are lock prefixed. The implementations use `Ordering::Relaxed` because the atomic elements themselves do not establish synchronization between tasks. This ordering guarantees atomic access to the element but does not establish a happens-before relationship for other memory. However, when using atomics the compiler is restricted in how much it can re-order, cache in registers, combine and most importantly vectorise Atomics [29, 9, 6]. Thus we incur the cost of losing a lot of compiler optimization.

An alternative approach that also stays within Velvet's constraints is to give each recursive leaf its own write buffer and merge these into the main data structure sequentially afterwards. This

avoids atomics entirely, so the inner loops remain ordinary sequential code that the compiler is free to vectorise and reorder. The cost is an allocation per leaf and a sequential merge phase whose runtime grows with the number of tasks, which by Amdahl's law bounds the achievable speedup. This was the initial approach taken for LBM before it was replaced by the atomic-based design. Also, to decrease buffer size, for the larger kernels global data structure index was computed during the sequential merge step rather then passing it in as a pair in the write buffer. This way one does not have to store a pair of (64bit, 64 bit) just one 64 bit value per write, yielding a big overhead decrease.

To conclude which overhead is larger, for every Velvet implementation with safe application level code two versions where created: One using Atomics and one using the merge structure. For a fair comparison Rayon also had its safe implementation extended to two versions.

6.2 Experience with Velvet and suggestions

```
(0..num_procs).into_par_iter().
   for_each(|myid| {
        let key_buff1 =
            unsafe { &mut std::
               slice::
               from_raw_parts_mut
               ((&ptr0).0, MAX_KEY
                as usize)[..] };
        let key_array =
            unsafe { &mut std::
               slice::
               from_raw_parts_mut
               ((&ptr1).0,
               SIZE_OF_BUFFERS as
               usize)[..] };

        let itrl = nb * myid;
        let mut itru = itrl + nb;
        if itru > NUM_BUCKETS as
           usize {
            itru = NUM_BUCKETS as
               usize;
        }
        for j in itrl..itru {
            let k1 = {
                if j > 0 {
                    bucket_ptrs[
                       myid][j -
                       1]
                } else {
                    0
                }
            };
            for i in k1..
               bucket_ptrs[myid][j
               ] {
                key_buff1[
                   key_buff2[i as
                   usize] as usize
                   ] -= 1;
                let k = key_buff1[
                   key_buff2[i as
                   usize] as usize
                   ];
                key_array[k as
                   usize] =
                   key_buff2[i as
                   usize];
            }
        }
    });
```

Under the right circumstances, Velvet is very easy to work with. When parallelizing a already recursive function, velvet makes it very easy by just annotating the function with #[spawnable]. However, when a sequential version has more then one parallelisable kernels 6 which need synchronization in between them. The function needs to be split up into multiple parts with a new velvet main caller function. This in and of itself is not hard to do, however when there are more functions like this it becomes a headache and requires extra code which otherwise would have not been needed[5].

A further addition would be to expose the two mutation strategies described in Section 6.1 as part of Velvet's API. In this thesis, both the atomic-backed shared buffer and the raw-pointer alternative had to be constructed by hand for every benchmark, even though the underlying pattern is identical across them: a contiguous output buffer that spawned tasks write into at disjoint indices. Providing this as a library type would remove a substantial amount of boilerplate and, more importantly, make the available options visible to the programmer rather than something each user must rediscover.

The two should be exposed differently, however. The atomic variant is safe and could be offered as an ordinary API. The raw-pointer variant is not, and shipping it as a default would reintroduce exactly the inherited unsafe code that Velvet is designed to avoid [18]. Offering it behind an explicit opt-in, such as a Cargo feature flag, would preserve that guarantee: the programmer who needs it must acknowledge the choice, while the default configuration remains free of unsafe code.

6.3 Performance Hypothesis

Liokouras et al. [18] found Velvet to be competitive with Rayon, Cilk and OpenMP on compute-bound divide-and-conquer kernels up to 128 cores. Velvet with unsafe and safe at application level is therefore expected to perform competitively with Rayon and C on the NPB EP benchmark, and to scale close to linearly, since EP decomposes into independently owned batches with only a small final reduction.

For IS: application-level unsafe code lifts aliasings restriction and is expected to recover part of the gap introduced by it, as it did for the merge sort benchmark of Liokouras et al. [18]. Therefore Velvet is expected to perform competitively with C and Rayon.

For LBM: the collide kernel is compute-bound and site-local, which fits Velvet's model well, whereas propagate moves populations across chunk boundaries and is memory-bound. Velvet is therefore expected to remain competitive with unsafe at application level. Velvet with application level safe code is expected to under perform C and Rayon because of the memory-bound propagate kernel.

Finally, the safe implementations allow a comparison between the two strategies described in Section 6.1. Atomics and per-leaf write buffers are expected to trade off against each other according to how much data a task writes relative to the work it performs. For EP, where each task produces only two sums and ten histogram bins, the merge buffers are small and the sequential merge is negligible, so the buffer-based version is expected to be faster: it retains vectorisation in the hot loop at almost no merging cost. For LBM, where every cell writes 37 populations, the

[5]This feature has now been added to velvet

buffers approach needs to write every result twice: once in the buffer and once during sequential merge. Thus incurring a significantly more memory traffic. The atomic version is expected to win despite the lost compiler optimisations.

7 Experiments

All experiments were conducted on the DAS-6 cluster [4] on the CPU:

Processor	2× AMD EPYC 7282 (Zen 2)
Cores / threads	32 cores, 64 threads (SMT2)
Clock frequency	2.8 GHz
L1 cache	32 KiB + 32 KiB per core
L2 cache	512 KiB per core
L3 cache	16 MiB per 4-core CCX (64 MiB per socket)
Memory	128 GiB DDR4, 2 NUMA domains
Scheduler	Slurm (exclusive node allocation)

The C versions of the NPB benchmarks (EP and IS, NPB 3.4.4) were compiled with GCC 12.4.0 at optimization level `-O3` with `-fopenmp` and `-mcmodel=medium`. The C reference of the SPEChpc LBM benchmark was compiled with GCC 12.4.0 using `-Ofast -march=native`, as specified in the SPEC configuration. Note that `-Ofast` enables fast-math optimizations, which relax IEEE 754 floating-point semantics: the C LBM baseline therefore runs at a more aggressive optimization level than the NPB C baselines and the Rust implementations. All Rust benchmarks were compiled with rustc 1.88.0 (stable) in release mode (`--release`), the Rust equivalent of an optimized build. The Rayon implementations use Rayon version 1.12.0 and the Velvet implementations use Velvet [17].

Source-code of the benchmarks run can be found at the authors github repository [5].

To evaluate performance the execution time will be compared. Each implementation will be ran on 1, 2, 4, 8, 12, 16, 20, 22, 24, 26, 27, 28, 29, 30, 31 and 32 number of threads with 10 iterations per thread count to calculate an average execution time per thread count number. Since our machine has 32 physical cores each with 2 threads this means that we wil only be utilising 1 thread per physical core.

The NAS Parallel Benchmarks define multiple standardized problem classes (S, W, A–F) 3.1. Larger classes are intended to stress modern parallel architectures and reduce the impact of fixed overheads [3].

The NPB EP implementations were evaluated using both the class C and class D problem sizes, corresponding to 2^{32} and 2^{36} random-number pairs, respectively.

The NPB IS implementations were evaluated using both the class C and class D problem sizes. Class C contains 2^{27} keys with a maximum key value of 2^{23}, whereas class D contains 2^{31} keys with

a maximum key value of 2^{27}.

LBM implementations where run on a grid of $X = 352 * Y = 1408$ for 10000 simulations following recommendations of benchmarking with lower grid size with more simulations from SPEC [28]

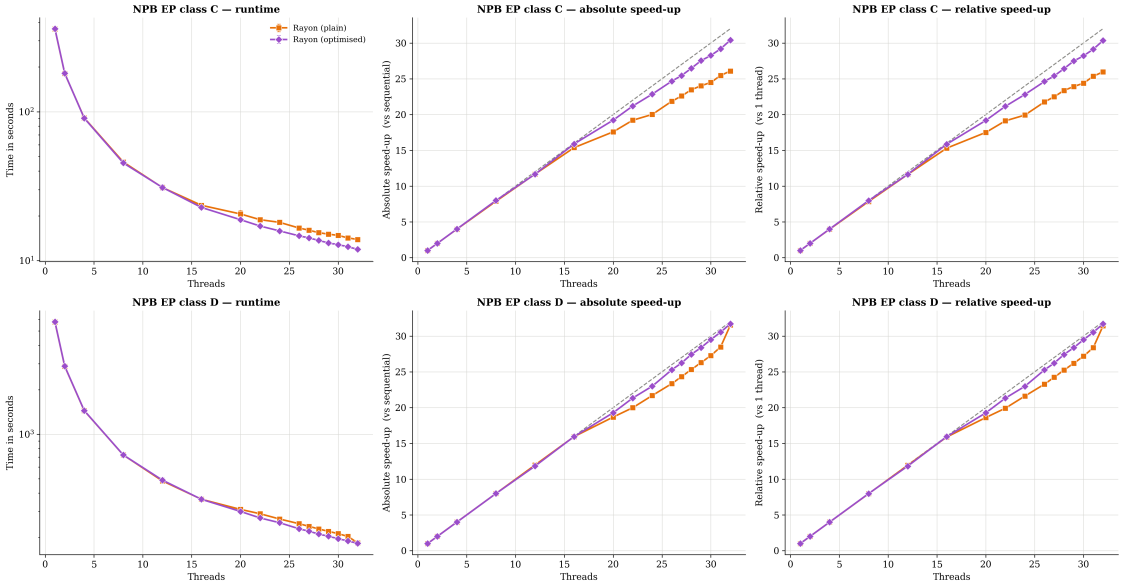
Before every iteration of every benchmark one untimed run is done to ensure all values are loaded in to memory. This is hard coded in every benchmarks source code. Of the whole program only the main benchmark loop is timed and later used for comparison.

Every benchmarks has its own verification process. Each implementation has exactly the same verification process and results are only considered usable if the verification check is passed.

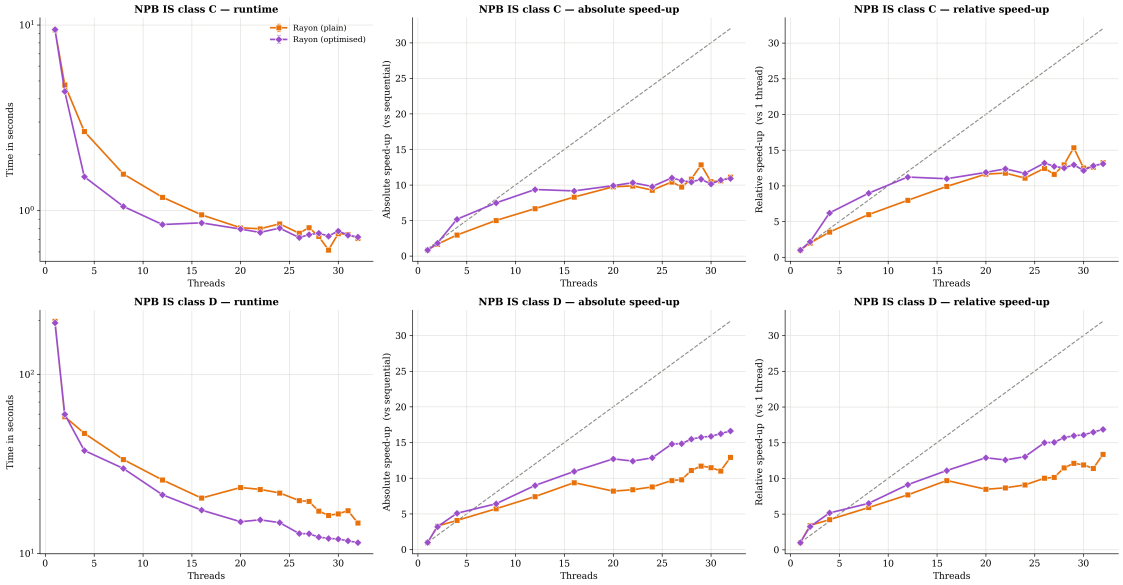
8 Results

8.1 Refactor of Martins et al

During the process of porting the already existing Rust NPB EP and IS source code of Martins et al [22] I recognized a pattern in the way work was distributed between threads. In both EP and in IS, all work was divided by the exact number of threads: every Rayon loop constructed exactly the same number of tasks as threads that where available. When one thread finishes its work and starts to look for work to steal, when there are just as many tasks as threads, there is no task to steal since all other task have been taken. Given that work stealing has been proven to be optimal [8]. I created a separate implementations where work was divided in more tasks then number of threads and evaluated both EP and IS.



(a) NPB EP, classes C (top) and D (bottom).



(b) NPB IS, classes C (top) and D (bottom).

Figure 1: Effect of over-decomposition on the Rayon reference implementations of Martins et al. [22]: the original one-task-per-thread decomposition (plain) against the over-decomposed version (optimised). Runtime, absolute speed-up (vs. sequential) and relative speed-up (vs. each variant's own 1-thread run); cf. Table 1.

Table 1: Effect of over-decomposition on the Rayon reference implementations of Martins et al. [22]: "Time in seconds" (mean over 10 runs) for the compute-bound EP and memory-bound IS kernels, classes C and D, comparing the original one-task-per-thread decomposition (plain) against the over-decomposed version (opt.) used throughout this evaluation.

	EP class C		EP class D		IS class C		IS class D	
Threads	plain	opt.	plain	opt.	plain	opt.	plain	opt.
1	360.2	361.1	5761	5777	9.395	9.433	197.5	193.6
2	181.0	181.5	2887	2894	4.759	4.380	58.00	59.75
4	90.93	90.55	1444	1447	2.668	1.521	46.84	37.55
8	45.94	45.30	723.2	723.6	1.573	1.053	33.42	29.79
12	31.00	31.03	482.8	488.8	1.180	0.8408	25.71	21.25
16	23.54	22.77	362.5	362.3	0.9483	0.8589	20.36	17.45
20	20.58	18.83	309.5	299.8	0.8078	0.7947	23.32	15.02
22	18.84	17.08	289.3	271.0	0.7977	0.7626	22.77	15.40
24	18.08	15.83	266.5	251.4	0.8484	0.8045	21.76	14.85
26	16.56	14.67	247.6	228.6	0.7549	0.7154	19.72	12.91
27	16.02	14.22	237.7	220.3	0.8092	0.7415	19.50	12.87
28	15.42	13.67	228.3	210.7	0.7269	0.7555	17.23	12.33
29	15.06	13.14	219.9	203.6	0.6129	0.7286	16.30	12.13
30	14.78	12.81	212.0	195.9	0.7515	0.7767	16.62	12.03
31	14.21	12.40	203.0	189.3	0.7447	0.7366	17.33	11.75
32	13.87	11.90	183.3	182.2	0.7089	0.7214	14.78	11.48

As can be seen in the graphs and table 1a, 1b, 1. In both cases the optimised versions with more task creation are faster in both smaller class C and D. For EP the optimised version performs the same form thread counts 1 till 12 after which the optimised version starts out performing the baseline and converges to a faster runtime on higher thread counts for both datasets. For IS on the smaller dataset C the optimised version outperforms the baseline by a great margin from threadcounts 4–12 after which they converge to each other with quite similar performances. On the bigger dataset D we see the opposite: as thread counts get larger both versions diverge from each other with the optimized version outperforming the baseline significantly.

Given both optimised versions outperform both baselines. For further comparison only the optimised versions will be used.

8.2 Application safe: Buffer versus Atomic approach

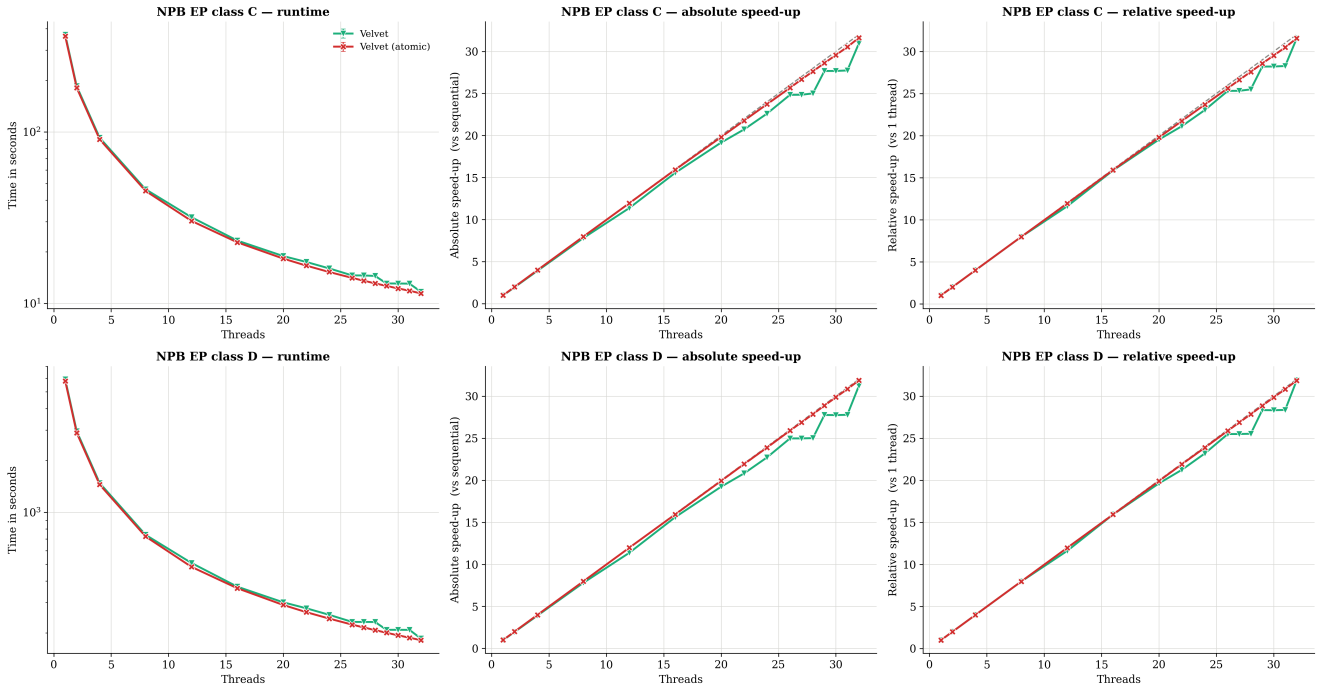
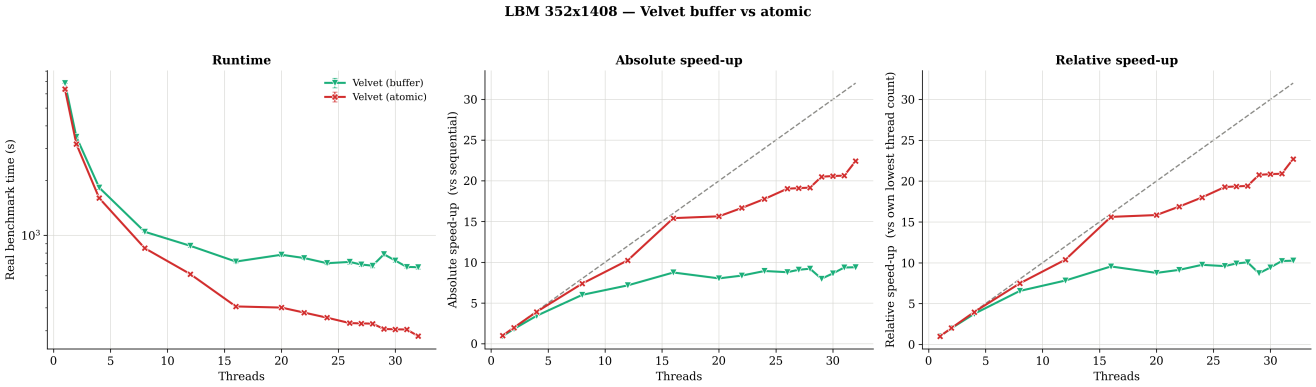
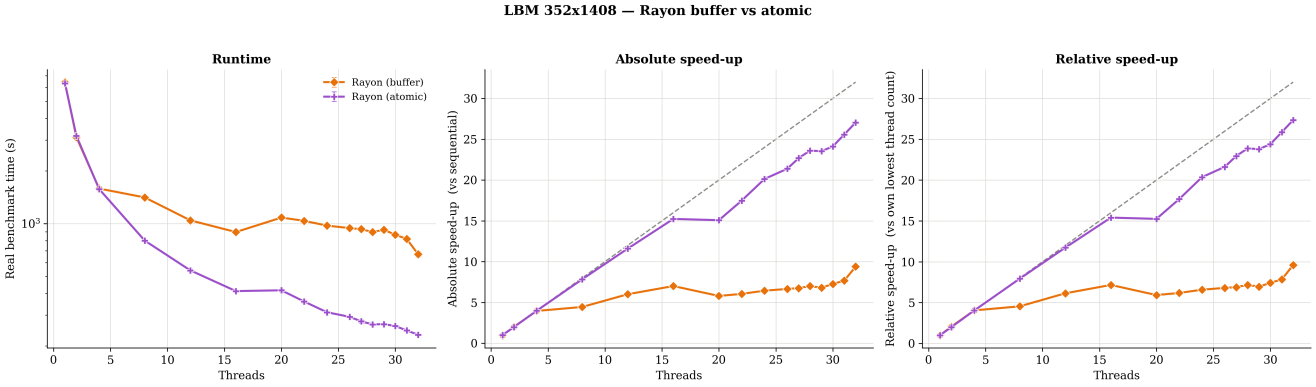


Figure 2: NPB EP classes C and D: Velvet with per-leaf write buffers vs. Velvet with atomic shared structures. Runtime, absolute speed-up (vs. the sequential implementation) and relative speed-up (vs. each variant's own 1-thread run).



(a) Velvet



(b) Rayon

Figure 3: LBM 352×1408: per-leaf write buffers vs. atomic shared structures for both frameworks.

Table 2: Buffer vs. atomic mutation strategy: mean time in seconds over 10 runs for every measured thread count. EP class D (Velvet) and LBM 352×1408 (Velvet and Rayon).

	EP class D (Velvet)		LBM (Velvet)		LBM (Rayon)	
Threads	Buffer	Atomic	Buffer	Atomic	Buffer	Atomic
1	5902	5779	6850	6354	6410	6345
2	2953	2891	3472	3175	3121	3179
4	1478	1447	1831	1603	1587	1577
8	740.2	724.3	1046	850.3	1412	800.5
12	508.4	482.7	876.1	612.7	1045	541.0
16	370.4	362.7	717.7	406.9	895.6	411.9
20	300.6	290.2	782.4	401.1	1083	416.1
22	277.8	263.7	750.5	376.5	1038	358.9
24	254.5	242.0	703.0	353.1	974.6	311.7
26	231.5	223.2	714.6	329.7	943.3	293.4
27	231.5	215.0	691.4	328.4	930.1	276.6
28	231.2	207.5	681.4	327.4	896.0	265.8
29	208.2	200.2	786.9	306.0	922.7	266.7
30	208.3	193.5	727.1	304.7	863.9	260.1
31	208.1	187.4	670.8	304.0	818.4	245.3
32	185.2	181.4	668.2	279.8	668.6	232.0

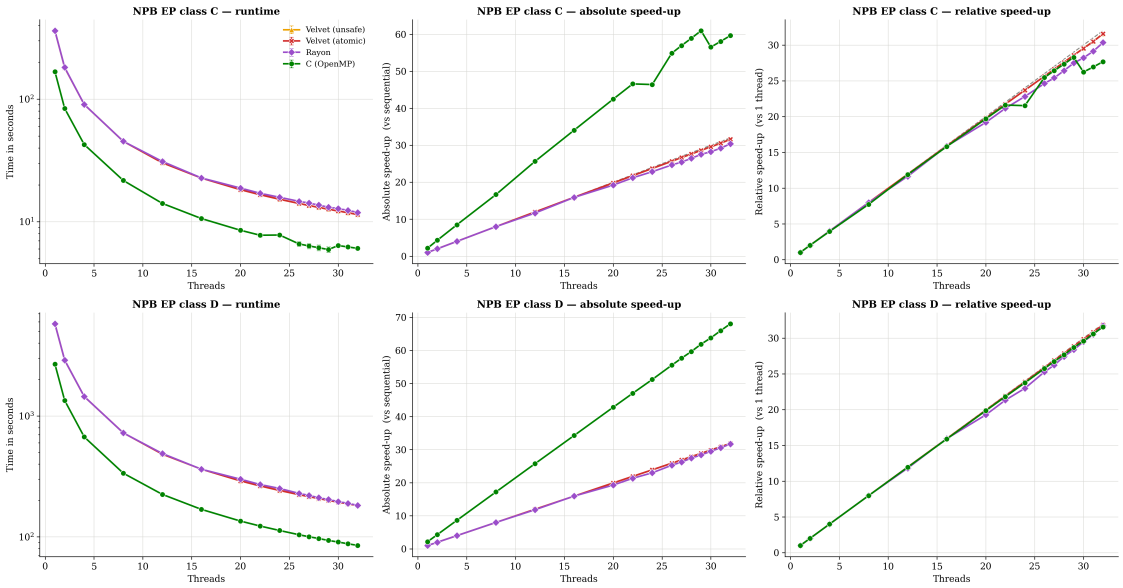
In EP: the gap on the single thread run (368.8 vs 361.2 s) 2 shows us that the atomic implementation does not pay a heavier overhead price for lost compiler optimisation compared to the buffering overhead. If it where, the buffer implementation should have outperformed the atomic implementation on single thread execution. This is the case because using atomics allows for an approach so that atomics are only used outside the hot loop, each batch accumulates into thread local sums and only touches the shared atomics at the final reduction. Given that atomics are only used outside of the hot loop: missed compiler optimisation is minimal. The buffer approach still has to merge results and thus incurs more overhead. Also The Velvet (buffer) variant shows a staircase between thread counts 26-31 2, 2. This is due task-count quantisation: with a limited number of leaf tasks T, runtime is governed by the busiest worker, T/p, which decreases only at particular thread counts. The atomic variant, with finer granularity, scales smoothly.

The hypothesis implicitly assumed atomics in the hot loop. The later discovered structure of the atomic implementation meant that that assumption did not hold so therefore it was initially wrong. However after new insights and results we can conclude that for EP Atomics has the clear upper hand and therefore we will only use atomics for later application safe evaluation.

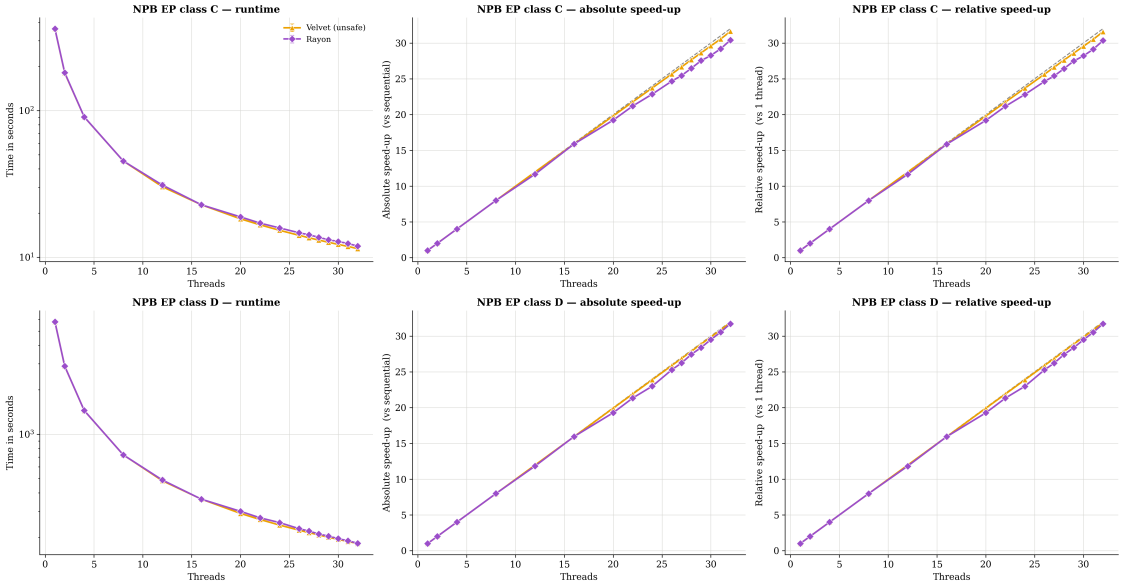
In LBM the hypothesis was right. The atomic implementation significantly out performs the buffer implementation. The expected merge/Amdahl effect becomes evident in the plateau shape: from a 6850 s base 2, capping at ≈670 s 2 means speedup saturates around 10×, implying a serial fraction of roughly 10%, which is approximately what a full-grid sequential merge per iteration would produce. The Rayon safe buffer and atomic implementation show the same pattern and

therefore strengthen the conclusion that atomic is also superior and will therefore be used as safe reference for later evaluation.

8.3 NPB: Embarrassingly Parallel



(a) Velvet (unsafe), Velvet (atomic), Rayon and the C (OpenMP) reference.



(b) Velvet (unsafe) against Rayon in isolation.

Figure 4: NPB EP, classes C (top row of each subfigure) and D (bottom row). Runtime, absolute speed-up (vs. the sequential Rust implementation) and relative speed-up (vs. each variant's own 1-thread run). All Rust variants coincide to within a few percent at every thread count, while the C reference is roughly twice as fast at every thread count: a gap already present at one thread, which is why its absolute speed-up exceeds the linear diagonal while its relative speed-up does not.

Table 3: NPB EP "Time in seconds" (mean over 10 runs) for classes C and D: Velvet (unsafe), Velvet (atomic), Rayon (over-decomposed) and the C (OpenMP) reference, against the sequential Rust implementation.

	Class C					Class D				
Threads	Velvet (unsafe)	Velvet (atomic)	Rayon	C (OpenMP)	Seq.	Velvet (unsafe)	Velvet (atomic)	Rayon	C (OpenMP)	Seq.
1	361.3	361.2	361.1	167.8	361.8	5780	5779	5777	2681	5781
2	180.6	180.7	181.5	83.91	–	2890	2891	2894	1341	–
4	90.46	90.47	90.55	42.64	–	1445	1447	1447	670.4	–
8	45.29	45.28	45.30	21.72	–	724.0	724.3	723.6	336.1	–
12	30.24	30.27	31.03	14.11	–	483.2	482.7	488.8	224.4	–
16	22.77	22.70	22.77	10.62	–	362.6	362.7	362.3	168.7	–
20	18.27	18.26	18.83	8.521	–	290.2	290.2	299.8	135.0	–
22	16.63	16.62	17.08	7.766	–	264.0	263.7	271.0	122.9	–
24	15.26	15.25	15.83	7.795	–	242.0	242.0	251.4	112.9	–
26	14.10	14.09	14.67	6.591	–	223.3	223.2	228.6	104.1	–
27	13.57	13.57	14.22	6.353	–	215.1	215.0	220.3	100.3	–
28	13.09	13.10	13.67	6.140	–	207.4	207.5	210.7	96.89	–
29	12.65	12.64	13.14	5.933	–	200.1	200.2	203.6	93.41	–
30	12.23	12.24	12.81	6.399	–	193.6	193.5	195.9	90.62	–
31	11.84	11.85	12.40	6.230	–	187.3	187.4	189.3	87.63	–
32	11.44	11.45	11.90	6.064	–	181.4	181.4	182.2	84.94	–

The Rayon optimised version is safe at application level, given the nature of the NPB EP hot loop and Rayon already splitting datastructures itself below API level: creating an unsafe implementation would be algorithmically the same.

The compute-bound EP kernel separates the results into two clear layers. Within the Rust field, all parallel variants are effectively indistinguishable: at 32 threads on class D, Velvet (unsafe), Velvet (atomic), and Rayon (optimised) finished within 0.5% of each other (181.4, 181.4, and 182.2 s), and Velvet (atomic) matches Velvet (unsafe) at every thread count. This confirms that the atomic-based safe design carries no measurable cost on this workload compared to is other unsafe counterparts. Thus satisfying the third property of "fearless" concurrency [1].

Table 4: Static instruction analysis of the compiled EP inner loops, columns as in Table 7.

		Hot loop				
Implementation	Codegen	pd	sd	256-bit	bounds	Verdict
Rayon	baseline (SSE2)	5	54	0	0	scalar
Velvet (unsafe)	baseline (SSE2)	5	64	0	0	scalar
Velvet (atomic)	baseline (SSE2)	5	54	0	7	scalar
C (OpenMP)	baseline (SSE2)	0	18	0	–	scalar

The C reference, however, outperforms every Rust variant by approximately a factor of two at every thread count, from one (2681 vs 5780 s on class D) through 32 (84.94 vs 181.4 s). This gap is not due to a difference in parallelism: the relative speed-up panels show all implementations, including C, scaling near-identically at ≈31.5–32× on class D, so parallelism is equivalent and the entire difference is inherited from the single-thread code generation. Static inspection of the compiled inner loops 4 excludes vectorisation as the cause: all four implementations, including the C reference (compiled `-O3` without `-march=native`), generate scalar code. The gap must therefore originate in the quality of the scalar code generated for the hot loop, dominated by the random-number generation and the logarithm and square root of the Marsaglia transformation, which GCC and rustc evidently compile with very different efficiency. Identifying the responsible construct is left as future work. Consequently, no conclusion can be drawn about the competitiveness of Rust with Rayon or Velvet against C on this benchmark. Note also that absolute speed-up is computed against the sequential Rust implementation, which C outruns twofold before any parallelism, so its curve sits at roughly twice the thread count: its relative speed-up is the meaningful scaling metric and is linear.

8.4 NPB: Integer Sort

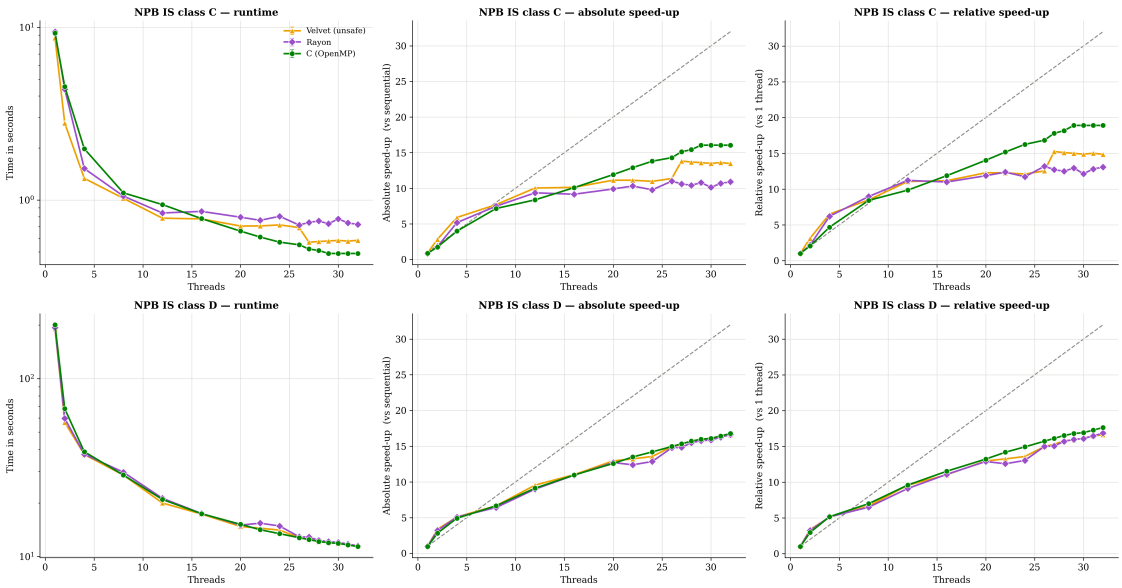


Figure 5: NPB IS, classes C (top row) and D (bottom row): Velvet (unsafe), Rayon and the C (OpenMP) reference. Runtime, absolute speed-up (vs. the sequential Rust implementation) and relative speed-up (vs. each variant's own 1-thread run); cf. Table 5.

Table 5: NPB IS "Time in seconds" (mean over 10 runs) for classes C and D. V (uns.) denotes the unsafe Velvet variant, Rayon the over-decomposed Rayon implementation, C the OpenMP reference, and Seq. the sequential Rust implementation.

	Class C				Class D			
Threads	V (uns.)	Rayon	C	Seq.	V (uns.)	Rayon	C	Seq.
1	8.677	9.433	9.26	7.866	191.3	193.6	200.7	190.9
2	2.784	4.380	4.53	–	56.82	59.75	67.72	–
4	1.335	1.521	1.98	–	37.47	37.55	38.81	–
8	1.023	1.053	1.10	–	28.70	29.79	28.70	–
12	0.7844	0.8408	0.94	–	19.99	21.25	20.93	–
16	0.7767	0.8589	0.78	–	17.33	17.45	17.41	–
20	0.7065	0.7947	0.66	–	14.75	15.02	15.17	–
22	0.7071	0.7626	0.61	–	14.44	15.40	14.15	–
24	0.7172	0.8045	0.57	–	14.07	14.85	13.44	–
26	0.6919	0.7154	0.55	–	12.78	12.91	12.76	–
27	0.5691	0.7415	0.52	–	12.50	12.87	12.45	–
28	0.5749	0.7555	0.51	–	12.12	12.33	12.15	–
29	0.5784	0.7286	0.49	–	12.03	12.13	11.95	–
30	0.5833	0.7767	0.49	–	11.83	12.03	11.86	–
31	0.5779	0.7366	0.49	–	11.64	11.75	11.62	–
32	0.5840	0.7214	0.49	–	11.50	11.48	11.38	–

On class D, the three implementations finish approximately 1% apart of each other at 32 threads: 11.38 s for C, 11.48 s for Rayon, and 11.50 s for Velvet (unsafe). And with an absolute speed-up of 16.6–16.8× against the 190.9 s sequential baseline 5.

Class C separates the field where class D does not. Rayon and C track each other relatively closely at low thread counts, while Velvet has a pronounced advantage between 2 and 12 threads: the three converge around 16 threads. After which the Rust variants flatten while C continues to improve, finishing at 32 threads in 0.49 s against 0.58 s for Velvet (19% ahead) and 0.72 s for Rayon (47% ahead), for relative speed-ups of 18.9× versus 14.9× and 13.1×. The most notable difference is that the parallel implementations on a single thread are significantly slower compared to the sequential version on class C and on class D perform almost the same. On class C, Velvet, Rayon, and the C reference take 8.68, 9.43, and 9.26 s against the 7.87 s sequential implementation (+10%, +20%, and +18%), whereas on class D the same implementations run within 0.2%, 1.4%, and 5.1% of sequential. The extra passes of the IS chunked algorithm: per-chunk histograms and a write-offset computation are real work on class C, but disappear on the fully bandwidth-bound class D. This overhead is why the absolute speed-up curves of Rayon and Velvet for class C sit visibly below C, while for class D the two panels nearly coincide.

Velvet additionally leads at low thread counts, most pronounced on class C at 2 threads (2.78 s against 4.38 s for Rayon and 4.53 s for C), narrowing to 3% by 8 threads and vanishing by 16. The advantage is already visible at one thread (8.68 vs 9.43 s), so it is not a scheduling effect. Given that IS is memory bound this could be due to the velvet implementation chunking data finer for

cache size. To verify this is future work.

8.5 SPEC: Lattice Boltzmann Fluid Dynamics simulation

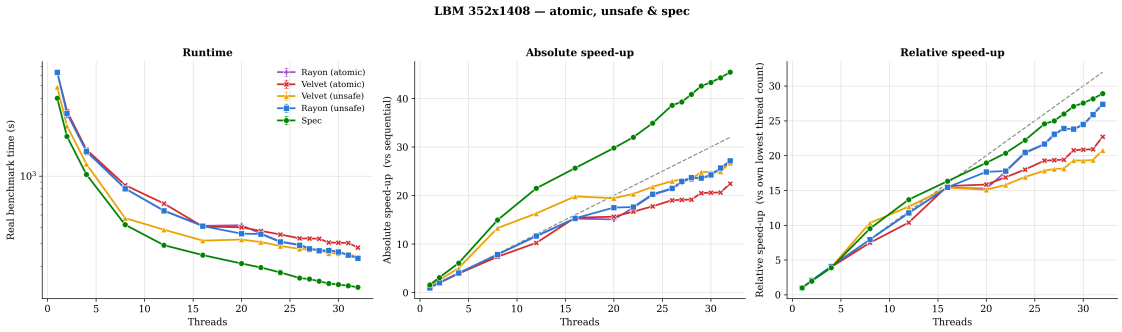


Figure 6: LBM 352×1408: all four Rust variants and the Spec reference. Runtime, absolute speed-up (vs. the sequential Rust implementation) and relative speed-up (vs. each variant's own 1-thread run); cf. Table 6.

Table 6: LBM 352×1408: real benchmark loop time (s), mean over 10 runs. V and R denote Velvet and Rayon, (at.) and (uns.) the atomic and unsafe variants, Spec the C reference (compiled `-Ofast -march=native`), and Seq. the sequential Rust implementation.

Threads	V (at.)	R (at.)	V (uns.)	R (uns.)	Spec	Seq.
1	6354	6345	4871	6328	3992	6274
2	3175	3179	2445	3046	2027	–
4	1603	1577	1240	1550	1030	–
8	850.3	800.5	472.1	799.1	420.1	–
12	612.7	541.0	384.7	537.8	291.9	–
16	406.9	411.9	316.6	410.0	244.6	–
20	401.1	416.1	322.5	358.3	210.6	–
22	376.5	358.9	308.7	356.0	196.1	–
24	353.1	311.7	287.7	309.4	179.7	–
26	329.7	293.4	273.3	292.2	162.6	–
27	328.4	276.6	269.6	273.9	159.7	–
28	327.4	265.8	268.5	264.7	153.6	–
29	306.0	266.7	252.3	265.8	147.4	–
30	304.7	260.1	252.6	258.6	144.9	–
31	304.0	245.3	251.5	244.3	141.7	–
32	279.8	232.0	234.6	231.0	138.1	–

The LBM benchmark combines a compute-bound collide kernel with a memory-bound propagate kernel, and the results reflect this duality: the ranking between implementations changes with thread count as the bottleneck shifts from computation to memory bandwidth. At one thread the field separates into three levels: the Spec reference at 3992 s, Velvet (unsafe) at 4871 s, and the two

atomic variants together with Rayon (unsafe) clustered at 6330–6360 s. At 32 threads it compresses differently: the Spec reference at 138 s, then Rayon (unsafe), Rayon (atomic) and Velvet (unsafe) converged at 231–235 s, and Velvet (atomic) at 280 s.

Table 7: Static instruction analysis of the compiled LBM collide loops. "Codegen" classifies the whole binary: AVX2+FMA indicates 256-bit packed and fused multiply–add floating-point instructions are present; baseline indicates only SSE2 scalar code. The hot-loop columns count packed-double (pd) and scalar-double (sd) arithmetic instructions in the collide region, how many packed operations use 256-bit registers, and the number of bounds-check calls (Rust only). All Rust variants were compiled without -C target-cpu=native; rebuilding with native code generation is left as future work.

		Hot loop				
Implementation	Codegen	pd	sd	256-bit	bounds	Verdict
Velvet (unsafe)	baseline (SSE2)	27	1335	0	12	scalar
Velvet (atomic)	baseline (SSE2)	16	1320	0	34	scalar
Rayon (unsafe)	baseline (SSE2)	22	1609	0	162	scalar
Spec (C)	AVX2 + FMA	311	163	144	–	packed SIMD

The performance gap of Rust with the Spec reference must be read in light of its compilation settings. As noted in the experiment section 7, the Spec binary is built with -Ofast and -march=native, where -Ofast enables fast-math optimisations that relax IEEE 754 semantics [13]: in the collide kernel, which performs 6606 floating-point operations per site, this permits the reassociation of floating-point operations, while -march=native additionally enables the AVX2 and FMA instruction sets. Static inspection of the compiled binaries confirms this asymmetry 7: the Spec binary is the only implementation containing packed 256-bit SIMD and FMA floating-point instructions, and its collide loop is dominated by packed operations, whereas every Rust variant compiles to scalar SSE2 code. Since none of the Rust binaries were built with -C target-cpu=native, it remains unknown how much of the Spec advantage would persist under equal instruction-set access. The comparison with the C reference is therefore confounded by the compilation asymmetry: the Rust variants come within (1.6×-2.1×) of a binary compiled with both fast-math and native SIMD, but how much of the remaining gap is attributable to the language rather than the compilation settings cannot be determined from these measurements and is left as future work.

Within the Rust field, the single-thread results isolate the serial cost of the safe atomic design. The Velvet atomic and unsafe variants differ only in their write mechanism, and the gap between them: 6354 vs 4871 s, approximately 30%, is therefore the direct measured price of storing the result grid as atomics. Static inspection of the compiled collide loops 7 rules out lost vectorisation as the mechanism: both variants compile to scalar code. The cost lies instead in the memory semantics of atomics themselves, which force every access to be performed as a memory operation and thereby prevent the register caching and instruction reordering discussed in Section 6.1. However, this cost is not symmetric across frameworks: Rayon (unsafe), at 6328 s, improves by only approximately 0.3% over Rayon (atomic) at 6345 s, whereas the difference between the two Velvet variants is

much larger. Static inspection shows that the Rayon unsafe collide region contains considerably more bounds-check sites than the Velvet unsafe region. However, these counts represent potential failure paths and their associated conditional branches, not calls that are necessarily heavy for performance. The static analysis therefore does not prove that bounds checking cancels out the benefit of removing atomics. It only identifies bounds-check-related control flow as a possible contributor to the remaining overhead. Confirming its actual impact would require dedicated kernel timing.

The scaling behaviour is governed by two hardware boundaries of the dual-socket node. First, every variant's performance stalls between 16 and 20 threads: Velvet (atomic) improves only from 406.9 to 401.1 s, and Rayon (atomic) regresses from 411.9 to 416.1 s. Precisely on the transition from one 16-core socket to two: pages which are first accessed on one NUMA node then need to be accessed remotely by the second socket's threads, while the first socket's memory controllers are already near saturation. Scaling then resumes better from approximately 22 threads as the second socket's memory bandwidth gets used more.

Second, at 32 threads Rayon (unsafe), Rayon (atomic) and Velvet (unsafe) converge to a common floor of approximately 230 s despite single-thread times 30% apart and Velvet (unsafe) being faster until $\approx$ 26 threads. Convergence of variants with different lower thread count performance to a shared ceiling indicates a shared bottleneck: the memory-bound propagate phase. At low thread counts the runtime is dominated by the compute-bound collide kernel, where Velvet (unsafe)'s faster serial code yields a large lead (472 vs 799 s at 8 threads). As collide time shrinks with parallelism: number of threads increase so partition size decreases, propagate's bandwidth demand becomes the limit for all variants equally. Experimenting with partition sizes at higher threadcounts to optimise this is added to future work. This also explains the apparent paradox that Velvet (unsafe) shows the weakest relative scaling of the Rust variants: relative speed-up rewards a slow baseline, and Velvet (unsafe) simply starts nearest the floor. Absolute runtime, not relative speed-up, is the meaningful comparison here.

Then, for the application level safe comparison, Velvet (atomic) tracks Rayon (atomic) exactly up to 16 threads, then falls behind by approximately 20% at 32 threads (279.8 vs 232.0 s). The difference therefore lies in how the two runtimes behave once the computation spans both sockets. The present measurements do not identify the responsible mechanism; a plausible candidate is Velvet's lock-based work queue, whose overhead Liokouras et al. [18] measured to be negligible for compute-bound workloads, but which has not been evaluated under the cross-socket, bandwidth-saturated conditions of this benchmark. Identifying if this is the cause can be done as future work. For the research question, the result stands on its own: the safe runtime costs nothing below 16 threads and approximately 20% at full core count on this workload.

8.6 Lines of code comparison

Table 8: Source size of the Velvet implementations against their Rayon baselines, per benchmark. *Lines of Code* counts non-blank, non-comment lines of the benchmark-specific source.

Benchmark	Implementation	Lines of Code	ΔLOC	Δ%
EP	Rayon (optimised)	246	–	–
	Velvet (atomic)	284	+38	+15.4
IS	Rayon (optimised)	570	–	–
	Velvet (unsafe)	791	+221	+38.8
LBM	Rayon (atomic)	2872	–	–
	Velvet (atomic)	3151	+279	+9.7

It can be concluded from the table 8 that except from the IS benchmark, all other LOC differences are rather small. This is due to the fact that the only addition velvet needed was the recursive wrappers of the compute functions. This was needed due to all Rayon kernels not being recursive already before porting to Velvet. The Velvet unsafe LBM implementation is not mentioned in the table due to it being approximately equivalent in LOC with the Velvet atomic version. If they would have, LOC differences would be even smaller. Also, given the new feature addition the IS metric is not valid anymore. The big difference in the IS kernel is due to the issues with a single source code function having multiple parallelisable kernels as described in the suggestion function 6.2. With the suggested function being implemented now this difference should drop a little. However, all these separate kernels still need recursive wrappers which yield extra code.

Besides the few extra lines of code, whom are just some basic recursive wrappers. During development we found Velvet actually being easier to debug then C and Rayon code. Using global read and write patterns made it easier to find common bugs one encounters when parallelizing code: the error could only be related to a few evident places in source code if parallelising is done kernel per kernel. Velvet also almost forces you to apply these global read and write methods because otherwise oversight is lost quickly! Applying different chunking sizes for data processing is also easier with velvet then with rayon as one just has to change the base case of the recursive function. With Rayon one has to sometimes alter the whole loop structure. Thus making Velvet easier to work with for cache usage optimising.

9 Conclusions

This thesis set out to answer the research question: *How does the performance and scalability of Velvet compare to the state-of-the-art?* To answer it, three benchmarks representing distinct workload types were ported to Velvet: the compute-bound NPB EP kernel, the memory-bound NPB IS kernel, and the mixed compute- and memory-bound SPEChpc LBM simulation, each was evaluated against Rayon, sequential Rust and the original C references on up to 32 cores.

On EP, all Rust variants finished within 0.5% of each other at 32 threads, and Velvet with application-level safe code matched its unsafe counterpart at every thread count. This confirms the

performance hypothesis for EP with respect to Rayon but not with respect to C, more importantly, satisfies the third property of fearless concurrency [1] on this workload: full compile-time safety was obtained without paying for it in performance and thus offering "fearless" concurrency on fully compute-bound kernels. On IS, the hypothesis that application-level unsafe code would make Velvet competitive was also confirmed: at class D, Velvet (unsafe), Rayon and the C reference finished within 1% of each other at 32 threads, with Velvet outperforming the state-of-the-art at low thread counts. On LBM, Velvet (unsafe) was competitive only throughout, converging with both Rayon application level unsafe and safe variants to a shared bandwidth-bound floor of approximately 230 s at 32 threads. The hypothesis for the safe LBM variant was only partially correct: rather than underperforming across the board, Velvet (atomic) tracked Rayon (atomic) exactly up to the 16-thread socket boundary and only then fell behind, by approximately 20% at full core count.

After benchmarking we can thus conclude that Velvet offers fearless concurrency [1] on fully compute bound kernels up to 32 threads something truly unique which Rayon: the Rust state-of-the-art for concurrent programming cannot offer.

The hypothesis comparing the two safe mutation strategies of Section 6.1 was confirmed for LBM but refuted for EP. For LBM the atomic design clearly outperformed the per-leaf write buffers, whose sequential merge phase capped speed-up at roughly 10×, approximately in line with Amdahl's law. For EP the buffer variant was expected to win, but the atomic variant proved equal or slightly faster at every thread count. However, the hypothesis had implicitly assumed atomics inside the hot loop, whereas the actual implementation only touches the shared atomics at the final reduction, so the anticipated loss of compiler optimisation never became reality while the buffer variant still paid its merge overhead.

Two comparisons could not be settled. Against C, the EP results are dominated by a factor-two difference in single-thread scalar code generation between GCC and rustc that is unrelated to parallelism, and the LBM comparison is confounded by the Spec reference being compiled with `-Ofast -march=native` while the Rust binaries received neither fast-math nor native SIMD code generation. In both cases the parallel scaling of the Rust implementations matched that of C, but no conclusion about the languages nor multi threaded execution performance against C can be drawn from these measurements. Similarly, since no fully safe Velvet implementation of IS was made, the fearless concurrency question remains open for purely memory-bound kernels.

In terms of usability Velvet outperformed Rayon. The easier debugging and cache optimization outweigh for the extra lines of code needed per benchmark (5-10 %).

In conclusion: for workloads that decompose into independently owned tasks, Velvet delivers on its promise of parallelism with no inherited unsafe code at no measurable performance cost and therefore offers "fearless concurrency". Where a workload requires heavy shared-memory communication, the price for application level safe Velvet code is paid in performance. However when dropping the application level safe code enables for competitive performance with C and Rayon. To draw conclusions about performance on memory and compute bound mixed benchmarks, more testing is needed. In terms of usability Velvet out performs C and Rayon due to its easier more oversight-full structure.

Considering all results velvet performs competitive with the state-of-the-art in the scope of this thesis.

10 Future Work

Several open questions arose throughout this thesis. First, the factor-two single-thread gap between the C and Rust EP implementations should be traced to the responsible construct in the hot loop, dominated by the random-number generation and the Marsaglia transformation, which GCC and rustc evidently compile with very different efficiency. Second, the LBM comparison with the Spec reference should be repeated with the Rust variants compiled with `-C target-cpu=native`, so that both sides have equal access to AVX2 and FMA instructions and the influence of the compilation asymmetry can be separated from that of the language. Third, the conclusions drawn from the static instruction analysis of the LBM collide loops rest on instruction counts alone, which show that certain instructions are present in the binary but not how much execution time they actually consume. Two attributions should therefore be verified with a runtime timer tool which measures where time is spent the Rayon (unsafe) bound check hot loops are indeed what cancels out its gain from removing atomics, and that the 30% serial gap between the Velvet variants is indeed caused by the memory semantics of atomic accesses.

On the memory side: the cause of Velvet (atomic) falling approximately 20% behind Rayon (atomic) beyond the socket boundary on LBM should be identified: a plausible candidate is Velvet's lock-based work queue under cross-socket, bandwidth-saturated conditions. Related to this, experimenting with partition sizes at high thread counts may raise the shared memory-bandwidth floor that all LBM variants converge to. For IS, the low-thread-count advantage of Velvet should be verified to have better cache locality due to finer chunking and if this can also be obtained by the other implementations in the same way. Finally, a fully safe Velvet implementation of IS remains to be built, so that the fearless concurrency question can also be answered for a purely memory-bound kernel, completing the workload spectrum covered in this thesis.

Considering benchmarking: the ported kernels should be run on bigger datasets. For EP and LBM the reason this thesis evaluated these smaller datasets was the lack of time. Running EP class D and LBM with current grid sizes and iteration numbers already took more than 16 hours to finish on available hardware. Due to various bug findings near the thesis publishing date there was simply not enough time left for bigger dataset evaluation.

References

[1] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. When is parallelism fearless and zero-cost with Rust? In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*, pages 27–40. Association for Computing Machinery, 2024.

[2] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*, pages 158–165. Association for Computing Machinery, 1991.

[4] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer*, 49(5):54–63, 2016.

[5] Quinten Max Baris. Npb-velvet: Benchmark implementations for the velvet thesis. https://github.com/QuintoniusB/thesis_final, 07-2026. Accessed: 2026-07-22.

[6] JF Bastien. No sane compiler would optimize atomics. Technical Report N4455, ISO/IEC JTC1 SC22 WG21, 2015.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216. Association for Computing Machinery, 1995.

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

[9] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 68–78. Association for Computing Machinery, 2008.

[10] Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti. Performance vs programming effort between Rust and C on multicore architectures: Case study in N-body. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10. IEEE, 2021.

[11] Will Crichton. The usability of ownership, 2020. Presented at HATRA 2020.

[12] David Drysdale. *Effective Rust: 35 Specific Ways to Improve Your Rust Code*. O'Reilly Media, 2024.

[13] Free Software Foundation. Using the GNU compiler collection (GCC), version 12.4.0: Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc-12.4.0/gcc/Optimize-Options.html, 2024. Accessed 2026-07-21.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 212–223. Association for Computing Machinery, 1998.

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018.

[16] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2nd edition, 2023.

[17] Badia Liokouras. Velvet: Parallel divide-and-conquer in safe rust. https://github.com/liokouras/Velvet.git, 07-2026. Accessed: 2026-07-22.

[18] Badia Liokouras, Ben van Werkhoven, and Rob V. van Nieuwpoort. Velvet: Parallel divide-and-conquer in safe Rust. In *Velvet: Parallel Divide-and-Conquer in Safe Rust*, 2026. To appear.

[19] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.

[20] Filippo Mantovani, Marcello Pivanti, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Performance issues on many-core processors: A D2Q37 Lattice Boltzmann scheme as a test-case. *Computers & Fluids*, 88:743–752, 2013.

[21] George Marsaglia and Thomas A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, 1964.

[22] Eduardo M. Martins, Leonardo G. Faé, Renato B. Hoffmann, Lucas S. Bianchessi, and Dalvan Griebler. NPB-Rust: NAS parallel benchmarks in Rust, 2025.

[23] Niko Matsakis, Josh Stone, and Rayon contributors. Rayon: A data-parallelism library for Rust. https://github.com/rayon-rs/rayon, 2026. Version 1.12.0, accessed 2026-07-20.

[24] Laura Moran and J. Mark Bull. Emerging technologies: Rust in HPC. Technical report, EPCC, The University of Edinburgh, 2023.

[25] Ricardo Pieper, Júnior Löff, Renato B. Hoffmann, Dalvan Griebler, and Luiz G. Fernandes. High-level and efficient structured stream parallelism for Rust on multi-cores. *Journal of Computer Languages*, 65:101054, 2021.

[26] Alex Rebert and Christoph Kern. Secure by design: Google's perspective on memory safety. Technical report, Google Security Engineering, 2024.

[27] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[28] Sebastiano Fabio Schifano, Luca Biferale, Enrico Calore, Alessandro Gabbana, Mauro Sbragaglia, Andrea Scagliarini, and Raffaele Tripiccione. 605.lbm_s (D2Q37): SPEChpc 2021 benchmark description. Standard Performance Evaluation Corporation (SPEC), 2021.

[29] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 306–316. Association for Computing Machinery, 2011.

[30] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.