Karlsruhe Institute of Technology

Improving Performance of Parallel Sample Sort for Nearly Sorted Inputs in Rust

Bachelor's Thesis of

Thomas Heinen

at the Department of Informatics
ITI - Institute of Theoretical Informatics, Algorithm Engineering

First examiner:	Prof. Peter Sanders
Second examiner:	Prof. Thomas Bläsius
First advisor:	Sascha Witt, M. Sc.

5. June 2023 – 5. October 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 5. October 2023

. . . .

Thomas Heinen

Abstract

Sorting algorithms are a fundamental part of computer science and play a crucial role in a wide range of applications. A desirable property for a sorting algorithm is to sort nearly sorted inputs faster than random inputs. This goal can be achieved by making the algorithm *adaptive*. An algorithm is called adaptive if it changes its behavior based on information obtained during the execution of the algorithm.

We present an approach that makes the sequential variant of the sorting algorithm IPS^{4}o adaptive. Our approach works by detecting specific patterns beforehand and switching to other algorithms that outperform the main algorithm on those patterns. On the pattern *unsorted-tail-1%*, which consists of a sorted sequence followed by random elements, our approach works particularly well: It runs over 6 times faster than the non-adaptive version of IPS^{4}o. We also apply our approach to the parallel variant of IPS^{4}o, where it is less effective but still works well on some inputs. For example, the *unsorted-tail-1%* pattern is 1.75 times faster than the non-adaptive IPS^{4}o.

In addition to algorithmic enhancements, we reimplemented IPS^{4}o in Rust, a modern programming language designed for performance and safety. We give a performance comparison between our implementation and the original implementation in C++. We point out important optimizations where *unsafe* Rust had to be used to make the Rust implementation competitive.

Zusammenfassung

Sortieralgorithmen sind ein fundamentaler Teil der Informatik und spielen in einer Vielzahl von Anwendungen eine wichtige Rolle. Eine wünschenswerte Eigenschaft für einen Sortieralgorithmus ist, dass er fast sortierte Eingaben schneller sortiert als zufällige Eingaben. Dies lässt sich zum Beispiel dadurch erreichen, dass man den Algorithmus *adaptiv* macht. Ein Algorithmus heißt adaptiv, wenn er sein Verhalten anhand von Informationen anpasst, die erst zur Laufzeit zur Verfügung stehen.

In dieser Arbeit präsentieren wir eine Möglichkeit, wie man die sequentielle Variante des Sortieralgorithmus IPS^{4}o adaptiv macht. Unser Ansatz funktioniert, indem wir vorab ausgewählte Muster in der Eingabe erkennen und gegebenenfalls andere Sortieralgorithmen verwenden, welche auf diesen Mustern schneller sind. Auf dem Muster *unsorted-tail-1%*, welches aus einer sortierten Folge gefolgt von zufälligen Elementen besteht, funktioniert unser Ansatz besonders gut: Unser Algorithmus ist mehr als 6-mal so schnell wie die nicht adaptive Variante von IPS^{4}o. Wir wenden unseren Ansatz auch auf die parallele Variante von IPS^{4}o an. Dort ist dieser zwar weniger effektiv, funktioniert auf einigen Mustern aber recht gut. Auf dem Muster *unsorted-tail-1%* ist der Ansatz beispielsweise 1.75-mal so schnell wie nicht adaptives IPS^{4}o.

Zusätzlich reimplementierten wir IPS^{4}o in Rust, einer modernen Programmiersprache, die ihren Fokus auf Performance und Sicherheit legt. Wir vergleichen die Performance unserer Implementierung mit der originalen Implementierung von IPS^{4}o in C++. Wir zeigen Optimierungen mit *unsafe* Rust auf, die notwendig sind, um näherungsweise so schnell zu sein wie die C++-Implementierung.

Contents

List of Figures

List of Listings

1. Introduction

Sorting is one of the most important and extensively researched problems in computer science. It is an integral part of many other algorithms, such as shortest path finding or common database operations. While the theoretical lower bound for a comparison based sorting algorithm on random data has long been proven to be $\Omega(n \log n)$ [1], *the one* fastest implementation does not exist. An efficient sorting algorithm must account for a wide range of different hardware features, including caching and branch-prediction. Additionally, for solving practical problems, not only random inputs are of interest, but also ones with pre-existing patterns, as those can be exploited to further speed up the algorithm. A simple example for such a pattern would be a sequence that is already sorted: We only have to check that this is indeed the case and are done. Some patterns are believed to be very common in practice and are thus worth optimizing for. An algorithm that exploits such input patterns is called *adaptive*.

The starting point of this work is IPS^{4}o (In-Place Parallel Super Scalar Samplesort) [2], a sorting algorithm based on samplesort. Samplesort is similar to the basic sorting algorithm quicksort: Where quicksort partitions the input into two parts, samplesort partitions it into many more. On random data, it performs better than most sorting algorithms used in standard libraries, most notably timsort [3] and pattern-defeating quicksort [4]. However, IPS^{4}o does not exploit input patterns at all, which makes it slow on some inputs compared to more adaptive algorithms such as timsort — despite being faster on random data.

The goal of this thesis is to alleviate that problem while still maintaining IPS^{4}o's excellent performance on random data. To achieve this, we look at different adaptive sorting algorithms and pick out the bits that fit our algorithm the best. These algorithms are described in Chapter 2. Additionally, we translated IPS^{4}o from C++ to Rust, a modern programming language designed for performance and safety. For a more detailed comparison between C++ and Rust, refer to Chapter 3. Lastly, to evaluate our findings, we measured our implementation across different patterns that are reasonable to expect in real world data. A description of our different adaptive approaches can be found in Chapter 4 and the corresponding measurements in Chapter 5. The reimplementation as well as the implementation of our adaptive approach can be found on GitHub [5].

2. Related Work

In this chapter, we go over different sorting algorithms that play a major role for this work. We start by explaining quicksort and samplesort, which build the foundation for many of the following algorithms. Secondly, we describe IPS^{4}o [2] in a more detail. This will later help us to understand why certain approaches did or did not work well. After that, we go over various other sorting algorithms that we tried to use to improve IPS^{4}o's adaptiveness.

2.1. Quicksort and Samplesort

Quicksort [6] is a rather simple recursive divide and conquer algorithm. First, a so-called pivot element is chosen. The pivot should be as close as possible to the median of the sequence, but selecting it should also be as cheap as possible. A simple, yet effective approach is to choose the pivot as a random element of the sequence. Afterward comes the partitioning phase, where most of the work is done. All elements smaller than (or equal to) the pivot are swapped to its left, all elements bigger than the pivot are swapped to its right. After partitioning, we recurse on both halves until each half only contains one element, which is trivially sorted. A simple optimization can be done here: We only recurse until a certain size and swap to another algorithm which is faster for small inputs, e.g., insertion sort. More advanced optimizations include parallelizing it [7, 8] or avoiding branch mispredictions [9].

Samplesort [10] is a generalization of quicksort where instead of only a single pivot, we choose multiple (and call them splitters). This results in not only two partitions per recursion step, but many more, which we call buckets. This is beneficial when sorting in parallel, because the subproblems can be solved independently of each other, enabling an easy parallelization.

2.2. IPS^{4}o

In-Place Parallel Super Scalar Samplesort (IPS^{4}o)[1] [2] is a sorting algorithm which is a faster in-place variant of *Super Scalar Sample Sort* (s^3-sort) [11]. As we reimplemented IPS^{4}o in its full form, we give a detailed overview on how the algorithm works. This also makes the reasoning behind our adaptive approach more comprehensible. While we tried to stay true to the original implementation, we did adapt some details. Most notably, when rounding bucket boundaries to block size (Section 2.2.3), we rounded *down* instead of *up*, which changes quite a few calculations in the permutation and cleanup phase — but the general idea stays the same. Rounding down has the advantage that we do not need an overflow bucket (cf. [2, Section 4.2]).

[1]The Latin word “ipso” means “by itself”, referring to the in-place feature of IPS^{4}o [2].

Our adaptive approaches in Chapter 4 mostly focus on optimizing the sequential variant of IPS^{4}o, so we do the same here. We explain the sequential algorithm in more detail and only point out the needed changes to go from the sequential to the parallel algorithm later.

The general algorithm can be roughly separated into four phases:
1. Sampling
2. Classification
3. Block permutation
4. Cleanup

2.2.1. Sampling

The algorithm starts by taking a sample of the input data. This is done by swapping `num_buckets` $\cdot \alpha$ randomly chosen elements to the beginning of the input. The original implementation chooses the oversampling factor $\alpha = 0.2 \log n$, where n is the length of the input. In our benchmarks, $\alpha = 0.25 \log n$ performed slightly better, so we adjusted it accordingly. A high quality of the splitters is very important for the algorithm to perform well, which is why a relatively big sample is chosen. Using random samples makes the algorithm simple and robust against malicious inputs. But, as we will see in Section 4.2, choosing the sample differently can be beneficial.

All sample elements are then swapped to the beginning of the sequence and sorted. Out of the sorted sequence, we pick `num_buckets - 1` splitters, equidistantly in the sorted sample. Swapping the sample to the start instead of auxiliary memory keeps the in-place property intact, while still allowing the oversampling factor to be dependent on the length of the input.

Out of our splitters, we build a binary search tree. This tree allows us to classify a given element into its correct bucket without conditional branching, which is an important factor in making the algorithm efficient.

2.2.2. Classification

We now have everything we need to start with the classification phase. For each bucket, we use an additional buffer block of size `BLOCK_SIZE`. When an element is classified into a bucket, it is first moved into the bucket's corresponding buffer. If this buffer is already full, we write the whole block back into the input. We know that there is enough space to do that, because at least `BLOCK_SIZE` elements had to be classified to fill the buffer. Figure 1 shows a typical situation during this phase. After classifying each element, our input is consists of full blocks followed by empty blocks. The missing elements are contained in the buffers. To retain our in-place property, we do not track which block belongs to which bucket. However, we do track how many elements we classified into each bucket because we need this information in the subsequent phases.

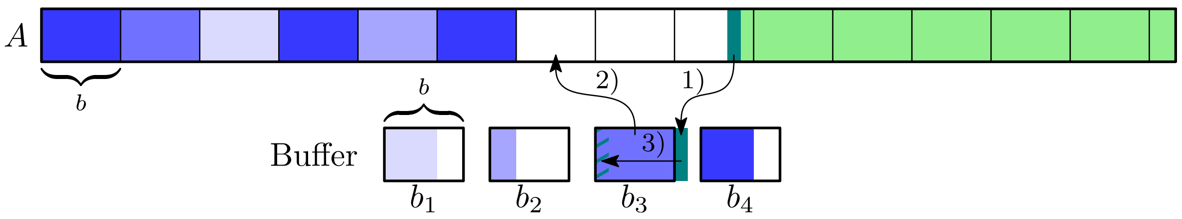


Figure 1: Classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. Here, the next element (in dark green) has been determined to belong to bucket b_3. As that buffer block is already full, we first write it back into the input, then write the new element into the now empty buffer. Figure and caption taken from [2].

2.2.3. Block Permutation

After all elements are classified, the blocks are likely to not be in the correct order. To fix that, we first determine the indices at which each bucket starts. This can be done by calculating a prefix sum over the number of elements that were classified into each bucket. For every bucket b_i, we now calculate two values, `write[i]` and `read[i]`. At the start of this phase, `write[i]` points to the beginning of bucket b_i, rounded down to block size. `read[i]` points to the first empty block of bucket b_i, or the end of the bucket if there are no empty blocks. The permutation upholds the following invariants for each bucket b_i:

- all blocks in b_i to the left of `write[i]` are correctly placed, i.e., they only contain elements belonging to b_i
- all blocks in b_i to the right of `read[i]` are empty
- all blocks between `write[i]` and `read[i]` are full, but have not yet been looked at and may need to be moved

For the permutation, we need an additional buffer with the size of two blocks, which we call *swap buffer*. We start at an arbitrary bucket, reading the block ending at `read[i]` into our swap buffer and decrementing `read[i]` by `BLOCK_SIZE`. We can determine to which bucket $b_{i'}$ the block belongs by classifying its first element.

If `write[i'] < read[i']`, we know that the block we want to write to is full. So now we copy that block starting at `write[i']` into our swap buffer and copy the previous contents of the swap buffer back to `write[i']`. We also increment `write[i']` by `BLOCK_SIZE`. To continue, we classify the block now contained in the swap buffer and repeat the process until we encounter an empty destination block to write to.

If `write[i'] ⩾ read[i']`, we know that the destination block is empty and can just write our block back at `write[i']`. We then go to our original bucket b_i and read our next block at the updated `read[i]`. If no block is left to read, that is, `write[i] ⩾ read[i]`, we go on to the next bucket. After doing this for every bucket, this phase is complete.

2.2.4. Cleanup

This phase starts with all blocks in their correct position and also some elements contained in the bucket buffers. After this phase, all elements will be placed correctly with regard to the splitters, and we can recurse into the buckets. Since the bucket boundaries do not necessarily align with the blocks, a block may be split by such a boundary. We call the partial block directly to the left of a bucket its *head*. The partial block at the end of our bucket is called *tail*.[1] The head only contains elements if at least one block was written back into our bucket. If that is the case, we copy all elements from the head to the beginning of the tail. We now only have to insert all elements from our buffers into the remaining tail. Because the tail of bucket b_i overlaps with the head of b_{i+1}, we have to process the buckets from right to left. Otherwise, writing elements into the tail of bucket b_i could overwrite elements in the head of bucket b_{i+1}.

2.2.5. Parallel Algorithm

To run IPS^{4}o in parallel, some additional work has to be done. Here, we only name the main differences between the parallel and the sequential version of IPS^{4}o, for a full explanation of the parallel algorithm refer to the original paper [2].

The sampling phase and building the classifier are done by the main thread. After that, the input is divided into as many *stripes* of about equal length as there are threads. Each thread can now classify all elements inside its stripe. Note that the classification phase is by far the most labor-intensive, so having this part fully parallelized results in a big speedup. Unfortunately, this phase can leave gaps of empty blocks in the middle of a bucket, which violates our invariant for the block permutation phase. We can restore said invariant by swapping the empty blocks to the end of the buckets they are currently residing in. Figure 2 depicts a typical situation after classifying all elements — the white, empty blocks of $t-1$ may need to be moved.

Most of the time, we only run the parallel algorithm for the first step of the recursion. By doing so, we already created many subproblems, each sorted concurrently with a sequential sorting algorithm, e.g., the sequential version of IPS^{4}o, IS^{4}o.

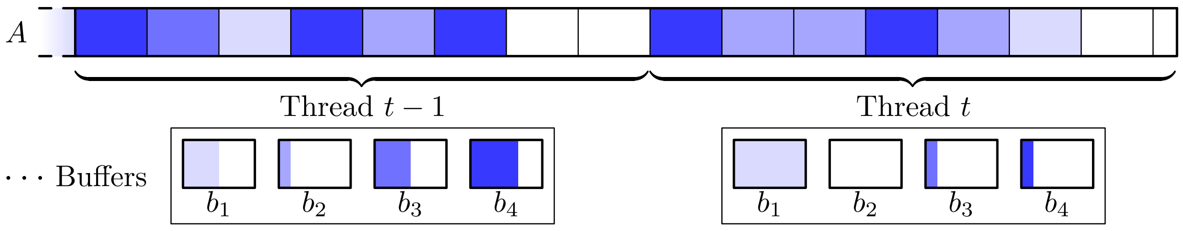


Figure 2: Input array and block buffers of the last two threads after local classification. Figure and caption taken from [2].

2.3. Timsort

Timsort [3] is an adaptive, stable sorting algorithm based on merge sort and insertion sort. It is named after its creator, Tim Peters, who first implemented it in 2002 for the Python programming language. Since then, it has become the standard sorting algorithm in quite a few languages, including Java, Swift and Rust [12, 13, 14].

[1]The contents of *head* and *tail* are different from the original paper because we rounded our bucket boundaries down instead of up.

The basic idea is to find presorted non-descreasing or strictly decreasing *runs* in the input and merging them. Descending runs are reversed before merging. That is the reason why only strictly decreasing runs are accepted, as to not reverse equal elements and to give up the stable property as a result. If a given run is too short, it is made longer by using insertion sort. Detecting runs is what makes timsort adaptive and, thus, interesting for us. A couple of other optimizations are made to make sure that merging is as efficient as possible, but they are of no consequence for our goals, so we skip them here.

2.4. Pattern-Defeating Quicksort

Pattern-defeating quicksort (pdq-sort) [4] is a sorting algorithm developed by Orson Peters. As the name suggests, it is based on quicksort. Like quicksort, it is also in-place and unstable. The original implementation was done in C++, but it is also implemented in Rust as part of the standard library [14, cf. `sort_unstable`]. The main algorithm is basically quicksort, but it applies many little optimizations that benefit its performance, quite a few of which we managed to use together with IPS^{4}o. Namely, performing optimistic insertion sort (Section 4.1), picking equidistant pivot candidates and falling back to random ones for bad partitions (Section 4.2), and pdq-sort handling of equal elements (Section 4.3.1). We will explain those optimizations in their respective section later on, so we don't go into further detail here.

2.5. Crumsort

Crumsort [15] is a sorting algorithm that is in-place, unstable and adaptive. It is a hybrid between a merge sort (quadsort [16]) and a quicksort variant, deploying a special partitioning scheme to make it especially efficient and branch-free. The main contributing factor to its adaptiveness is the `crum_analyze`-function, where the input is scanned for its "sortedness". If the input is nearly sorted, the input or parts of it are sorted with quadsort and unsorted parts are sorted with the quicksort based approach. If the input was sorted using both quadsort and quicksort, the resulting sorted parts are merged in-place. While the original implementation is in C, there also exists a reimplementation [17] in Rust. The Rust implementation does not contain the `crum_analyze`-function. However, it does contain a reimplementation of quadsort and the in-place merger.

2.6. Glidesort

Glidesort [18] is a sorting algorithm that, like timsort, is both stable and adaptive. According to its author, it combines the best-case behavior of timsort-style merge sorts for pre-sorted data with the best-case behavior of pattern-defeating quicksort for data with many duplicates. Glidesort builds on the foundations of timsort and powersort [19], which uses heuristics to find a good order for merging sorted runs. Unlike timsort, which eagerly sorts unsorted subsequences, glidesort defers the sorting as long as possible. This results in longer unsorted subsequences, on which (stable) quicksort (inspired by fluxsort [20]) can be used. Another important part of glidesort for our purposes is its in-place merger, which we need for our `crum-analyze`-function in Section 4.3. We go into more detail on how the merger works in Section 4.3.3.

3. Case Study: From C++ to Rust

In this chapter, we will first explain what makes Rust a desirable language to implement sorting algorithms in, and go on to describe the difficulties and insights we gained during our translation from C++ to Rust. We will also provide measurements on how the two implementations compare performance-wise.

3.1. Rust

Rust is a statically typed, general-purpose language, declaring itself as "empowering everyone to build reliable and efficient software".[1] It was initially developed by Graydon Hoare as a part-time side project in 2006, and after the language gained some maturity it got sponsored by Mozilla [21]. As already teased in its slogan, Rust has two main selling points: efficiency and reliability. These two points line up perfectly with what we want from a basic algorithm such as sorting. Efficiency comes with the fact that Rust does not rely on the use of a garbage collector, which saves a lot of possible runtime overhead. Despite the absence of a garbage collector, Rust is still memory-safe, preventing a whole class of bugs [22], and giving it an edge over other efficient languages such as C and C++. This combination of memory-safety without runtime overhead makes Rust stand out and is facilitated by introducing a notion of *ownership* into its type system [23]. Another side effect of the ownership system is that data races are completely avoided[2].

In Rust, every object has a unique owner. The owner can pass ownership to a new owner, and it can also lend out references to or into the object. These references, also called *borrows*, are confined to the current scope, meaning that there can not be an outstanding reference (otherwise known as a "dangling pointer") when the owner goes out-of-scope. As soon as the owner goes out-of-scope, the owned object can safely be deallocated.

An additional feature of Rust is its distinction between *safe* and *unsafe* code: Safe code cannot exhibit undefined behavior and is memory safe — assuming no compiler bugs or faulty libraries that use unsafe code. In contrast, in unsafe code, the programmer gains more flexibility, but also has the obligation to uphold certain invariants themselves, as to not trigger undefined behavior. It is possible to build safe abstractions around unsafe code, so the unsafe code is well contained. However, safe code may prevent one from writing perfectly efficient programs. For example, writing to and reading from an uninitialized array is not allowed in safe Rust, thus one has to live with the overhead of array initialization, or use unsafe code. Another example is bounds checking when accessing

[1]Website: https://www.rust-lang.org/

[2]Race conditions and deadlocks are still possible.

elements inside a slice[1]: By default, Rust ensures that all slice accesses are in-bounds. If that is not the case, the program crashes instead of exhibiting undefined behavior. In many cases, the compiler can reason that no accesses will be out of bounds and no actual runtime check is needed. Listing 1 shows an example where the compiler is usually smart enough to figure out that `i` is always in-bounds because the loop condition already ensures exactly that.

```rust
let v: Vec<i32> = Vec::new();
// fill v with values
for i in 0..v.len() {
  let current = v[i];
  // do something with "current"
}
```

Listing 1: Repeated slice access for which the compiler can determine that bounds-checks are unnecessary.

3.2. Use of `unsafe`

While we restricted our implementation to only sort types that implement the `Clone` and `Default` traits to cut down on unsafe-usage, we still didn't fully get around using it without taking a notable performance hit. We ended up using unsafe code in two[2] places:

3.2.1. Inserting Elements into Buffers

Although the Rust compiler is very good at optimizing and removing unnecessary bounds-checks, it will not catch every possible way of avoiding them while still maintaining memory safety. Even when the compiler fails to remove bounds-checks, they oftentimes do not come with a noticeable performance loss, as only very few additional instructions are introduced, and branch-prediction will probably always predict the correct path. But in hot code, every instruction counts. With IPS⁴o, this is the classification phase, as the related code is called at least once for every element. For the actual classification, that is, assigning the bucket number to an element by traversing the search tree, the compiler is smart enough to reason that no explicit bounds-checks are necessary. In contrast, when we then insert the classified element into its corresponding buffer, the compiler cannot reason about the correct handling of indices. As a consequence, the compiler introduces explicit bounds-checks to ensure memory safety. To circumvent this, we replace the `push` method on our `BucketBuffer` with `unchecked_push`, as seen in Listing 2. The new method, `unchecked_push`, places the obligation on the caller to only push (up to) `BLOCK_SIZE` elements into the buffer. Note that the obligation doesn't really change, but if it is violated, the program now exerts undefined behavior instead of crashing. If we want to push more elements into the buffer, we have to clear it first by writing it back into the input slice. Violating this agreement would cause the program to write to a memory location outside the given slice, instantly causing undefined behavior.

[1]A slice is a reference to a contiguous piece of memory, e.g., an array.

[2]Strictly speaking, we also use unsafe code in our implementation of insertion sort, but since we just took the implementation from the standard library, we will not go into detail about that.

```rust
pub fn push(&mut self, elem: T) {
  self.bucket[self.len] = elem;
  self.len += 1;
}
```

```rust
pub unsafe fn unchecked_push(&mut self, elem: T) {
  *self.bucket.get_unchecked_mut(self.len) = elem;
  self.len += 1;
}
```

Listing 2: Optimizing bucket-insertion by eliminating bounds-checks. On the left we see the unoptimized and on the right the optimized version.

3.2.2. Initialization of Buffers

Reading from uninitialized memory can lead to undefined behavior, so Rust simply disallows it. Additionally, the Rust compiler assumes that all types are properly initialized for optimizing purposes. These two properties lead to the obligation to initialize every memory location we read from, at least in safe code. Similar to bounds-checks, the compiler could theoretically optimize the initialization, but it can realistically only do so in very simple cases — and not in our case. Since the buffers that IPS^{4}o uses are quite large (multiple hundred kilobytes), initializing them comes at a noticeable cost, especially for small inputs. To circumvent this issue, we can use the type `MaybeUninit<T>`[1] to tell the compiler that the contained type `T` may not be initialized. This allows us to only initialize the buffer element by element as we copy the values into the buffer during classification. Since we count how many elements we copied into the buffer, we can be sure that exactly this many values are initialized and thus safe to read when copying the values back into the input slice. Figure 3 depicts the running times of the algorithm before and after the mentioned optimizations.

```rust
pub struct BucketBuffer<T: Clone> {
  bucket: [T; BLOCK_SIZE],
  len: usize,
}

pub fn push(&mut self, elem: T) {
  self.bucket[self.len] = elem;
  self.len += 1;
}

pub fn clear(&mut self) {
  self.len = 0;
}

pub fn get(&self) -> &[T] {
  return &self.bucket[0..self.len]
}
```

```rust
pub struct BucketBuffer<T: Clone> {
  bucket: [MaybeUninit<T>; BLOCK_SIZE],
  len: usize,
}

pub fn push(&mut self, elem: T) {
  self.bucket[self.len].write(elem);
  self.len += 1;
}

pub fn clear(&mut self) {
  for i in 0..self.len {
    unsafe {
      self.bucket[i].assume_init_drop()
    }
  }
  self.len = 0;
}

pub fn get(&self) -> &[T] {
  unsafe {
    return &*(&self.bucket[0..self.len]
      as *const [MaybeUninit<T>]
      as *const [T])
  }
}
```

Listing 3: Optimizing `BucketBuffer` by avoiding unneeded initialization. On the left we see the unoptimized and on the right the optimized version.

[1]https://doc.rust-lang.org/stable/std/mem/union.MaybeUninit.html

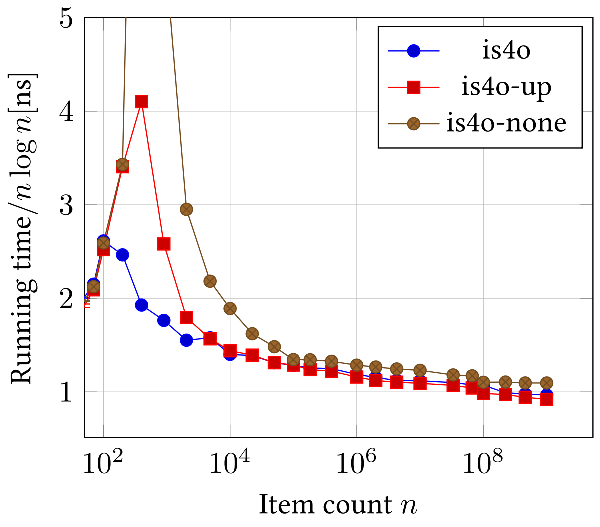


Figure 3: Running time across the `unsafe` optimizations described in Section 3.2, **is4o-none**: before unsafe optimizations, **is4o-up**: only `unchecked_push` optimization, **is4o**: optimized sequential implementation.

3.3. Additional Notes

We noticed some additional differences between Rust and C++, but they do not warrant individual sections for each of them.

Mutable Slices: The parallel version of IPS^{4}o requires giving each thread a part of the input slice and sorting it independently of the others. This requires the slice to be split into multiple mutable slices. Because we want the slices to be aligned to the block size and want to balance their lengths the best we can, the slices are possibly of varying lengths. In C++, we just give the whole slice to every thread and only have to ensure that the threads access disjunct subslices. In safe Rust, this approach is not possible, as there would be multiple mutable references to the same value at the same time, which is not allowed. While something similar to C++ can surely be achieved with the help of unsafe Rust, we opted for the safer approach and use `split_at_mut` multiple times, storing the resulting mutable slices in a `Vec`. One downside of this approach is that we need an allocation for our `Vec`. Note that we can't simply use an array instead of a `Vec`, as we do not know the number of slices during compile time.

Thread Pool and Lines of Code: The C++ implementation uses `OpenMP`-threads, and if they are not available, it uses a custom thread pool implementation. We decided to rely only on the global `ThreadPool` of `rayon`[1], without implementing our own thread pool. This saves us around 300 lines of code. In total, the C++ implementation has ≈ 2400 LOC, while the Rust implementation has ≈ 2000 LOC, making the Rust implementation 400 LOC or about 15% shorter.

Rust's Sorting Safety: Rust has stricter requirements than C++ has for a sorting function to be a fill-in replacement for the standard library sorting algorithm:

- if a user-defined comparison function does not implement a total order, the algorithm should terminate and the slice should contain all original elements, in unspecified order

[1]https://github.com/rayon-rs/rayon

- if the comparison function panics (equivalent of a C++ exception), the slice should contain all original elements, in unspecified order
- if the type has interior mutability, every modification incurred by calling the user-defined comparison function should be visible, even if the comparison function panics

Neither the Rust nor the C++ implementation of IPS^{4}o fulfills those stricter requirements. If the sorting algorithm uses `unsafe` mem-copies to move elements, the first two points are required to retain memory safety. We are confident that our implementation can be changed to fulfill these requirements.

Unstable Features: As Rust is still a relatively new language, some desirable features are missing. It is possible to use features of Rust that are not yet stabilized (and are not guaranteed to ever be stabilized). In most cases it is possible to achieve the same results "by hand", but to increase iteration speed we make use of some of those features. An example of a function that is locked behind an unstable feature flag is `is_sorted`, which checks if a slice is sorted. When using unstable features, it is required to use a nightly toolchain, which in our case is `nightly-2023-01-30`. For a "production-ready" version of our implementation, it would be desirable to use a stable Rust version.

3.4. Performance

Despite applying various optimizations, we did not manage to get our Rust implementation to the same performance as the C++ implementation. Figure 4a depicts the running times of both sequential implementations on uniformly distributed 32-bit integers. The measurement was taken on an *AMD Ryzen 9 3950X* CPU with 64 GB of RAM. We compiled the C++ code with GCC 10, with opt-level 3. The Rust implementation is about 2.5% faster for input sizes between 200 and 2048 elements. For larger sizes, upward of 2048 elements, it is 7.6% slower, on average.

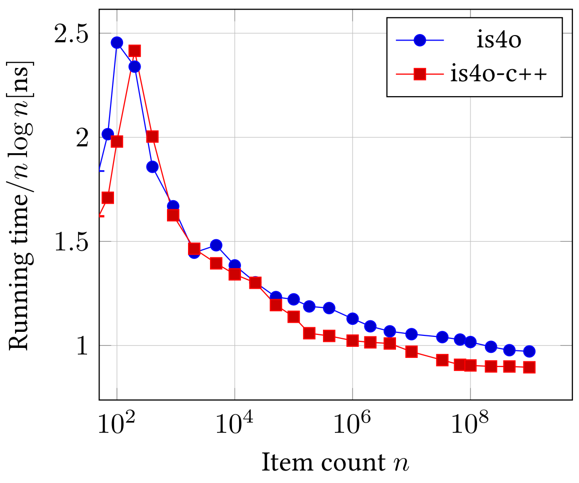
As the parallel version uses the sequential algorithm to do most of the work, it is unsurprising that the parallel Rust implementation is slower as well. For inputs of size $50,000$ or bigger, the C++ implementation is 16.7% faster. For smaller inputs, both implementations use only the sequential algorithm, which we already compared above.

We do not know if the reason behind this performance gap is some missed optimization on our part, or if the Rust compiler emits slower code than the C++ compiler does. To make the comparison as fair as possible, we also tried compiling the C++ code with Clang. Clang uses the same compiler backend as the Rust compiler, which is LLVM[1]. However, changing to Clang doesn't really change the performance of the C++ implementation, so it still remains faster than our Rust implementation. We did not focus on optimizing the parallel version, because it is probably impossible to catch up to the parallel C++ implementation without first having the sequential Rust implementation be as efficient as the C++ one.

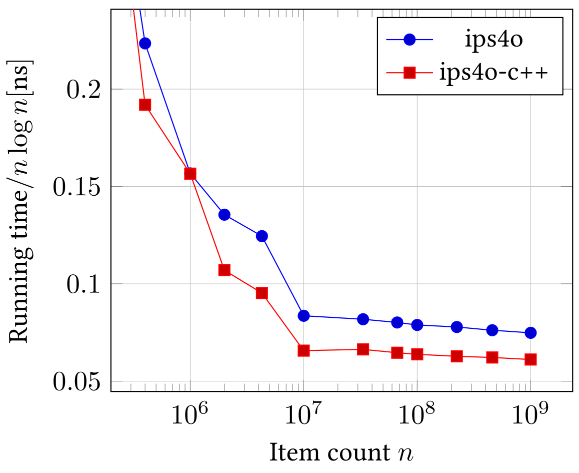
We did notice that both the crumsort and the quadsort implementation [17] were slower than their C counterparts [15, 16]. Although we did not investigate this further and cannot be sure that the implementation is of a high quality, this is an indication that Rust performs

[1]https://llvm.org/

worse than C and C++ in the realm of sorting algorithms. Further work is needed to confirm this suspicion.



(a) Comparison of sequential algorithms



(b) Comparison of parallel algorithms

Figure 4: Performance comparison of the Rust and C++ implementations of IS^{4}o and IPS^{4}o. Both measurements were taken on an *AMD Ryzen 9 3950X* with 64 GB of RAM.

4. Adaptive Approaches

In the following chapter, we describe different optimizations that can be used to make a sorting algorithm more *adaptive* [24], which means that it takes advantage of presortedness of the input. We explain which existing algorithm each optimization is inspired by, and how we adjust them to work together with our implementation of IPS^{4}o. In order to test these different approaches, we measure their performance impact across a range of different input patterns. We assume that these patterns are likely to occur in real world data and are thus worth optimizing for. The results of our measurements and precise explanations of the patterns are presented in Chapter 5.

4.1. Partial Insertion Sort

Partial insertion sort was first described in literature by Orson Peters in his paper about pattern-defeating quicksort [4] (cf. Section 2.4). Peters used it in pdq-sort after seeing it in Howard Hinnants `std::sort` implementation of Clang.

Listing 4 shows the pseudocode for the partial insertion sort procedure. The idea is to find up to `MAX_STEPS` (a fixed number) of out-of-order elements and sorting them using insertion sort. If the insertion sort finishes before moving `MAX_STEPS` elements, the input is fully sorted, and we are done. This also covers the case of an already sorted input, as there are no elements out of place and the procedure can exit without moving any elements. If the insertion sort did not finish before moving `MAX_STEPS` out-of-order elements, the sequence is sorted with the main algorithm, in our case that's IPS^{4}o.

Relative to the main sorting algorithm, the overhead of running the partial insertion procedure *once* at the start is minimal. So if less than `MAX_STEPS` out-of-order elements exist, the speedup is enormous, but if there aren't, there is no notable negative impact. However, it doesn't provide any meaningful progress to our main sorting algorithm, so we still shouldn't run partial insertion sort *every* step of the recursion. To solve this issue, a heuristic can be applied to only use partial insertion sort when the sequence is assumed to be nearly sorted. Pattern-defeating quicksort uses the following heuristic, which consists of two parts, and if both of them apply, partial insertion sort is used:

1. In order to choose a pivot, multiple pivot candidates are chosen. If all of them are in the correct order already, it is likely that the input as a whole is sorted as well.
2. If no element was swapped during the previous partitioning phase, it is highly likely that the input is (nearly) sorted.

We integrated partial insertion sort into our algorithm using the same heuristic as pattern-defeating quicksort. However, we did not see any improvements for any of our patterns, so we decided not to include it for the end result of this thesis. Future work could easily integrate partial insertion sort as an additional optimization during preliminary pattern detection, for example, if a chunk was detected as partition-friendly (see Section 4.3).

```
input: $v$: mutable slice of length $n$
output: boolean $b$, denoting if the slice $v$ is now sorted

MAX_STEPS ← 8
$i \leftarrow 1$
repeat MAX_STEPS times:
    // Find next pair of out-of-order elements.
    while $i < n$ and $v[i-1] \leq v[i]$:
        $i \leftarrow i + 1$
    end
    // Check if everything is sorted.
    if $i = n$:
        return true
    end
    // Shift the $i^{\text{th}}$ element to the correct position in
    // the sorted sequence left of $i$.
    shift_left($v$, $i$)
end
return false
```

Listing 4: Partial insertion sort

4.2. Equidistant Sampling

IPS^{4}o chooses its splitters by swapping `s` randomly chosen elements to the beginning of the array, where `s = num_buckets` $\cdot \alpha$, with α being the oversampling factor, see Section 2.2.1. These elements are then sorted, and the splitters are picked equidistantly from the sorted sequence, which leads to a high probability for good splitters.

In contrast to that, equidistant sampling already picks the sample from the input in an equidistant manner. Like with random sampling, the sample now has to be sorted, and the splitters are picked equidistantly from the sorted sample. Although the difference of both methods isn't very big, there are two things equidistant samples do better: First, we expect equidistant samples to lead to a higher quality selection of splitters for non-malicious inputs. Especially for cases where a big part of the input is already in order, choosing samples equidistantly results in nearly perfect splitters. Note that for random inputs, the quality of splitters from equidistant and random sampling is exactly the same. Second, equidistant sampling is more cache friendly and additionally allows the CPU to do prefetching as the indices are predictable.

A downside of equidistant sampling is that it is completely deterministic and, thus, susceptible to malicious inputs. To avoid this issue, we adapt a method from pattern-defeating quicksort [4]: When we encounter a bad partition, we break patterns in the input sequence by picking the samples randomly again in the next recursion. Detecting a bad partition is easy — if one bucket is much bigger than the others, the partition is bad, and we fall back to random sampling. This optimization was successful and the results are presented in Section 5.2.

4.3. Preliminary Pattern Detection

Partitioning based sorting algorithms like samplesort don't excel at exploiting patterns in the input sequence. One way to perform better on certain patterns is to pick good pivot candidates, for example by picking samples equidistantly, as described above. But for many patterns, this is not enough to compete with merge-based algorithms, which can take advantage of saw-like patterns much more naturally.

A relatively simple way to get around that problem is by searching for patterns before starting with the partitioning phase. The search for patterns has two main criteria that should be fulfilled: For one, the most common patterns should be found, but, just as important, the pattern detection needs to be quick. This part is crucial for the case that there is no pattern to be found. A good compromise for us was to adapt the `crum_analyze`-function implemented in crumsort [15] (see Section 2.5) and other algorithms by the same author. The idea is to split the input into multiple parts, for example eight, and to scan each of these parts independently. In the following, we will call these parts "chunks"[1]. During this scan, there are five different cases that can occur per chunk: sorted, reversed, partition-friendly, merge-friendly, and unsorted. After scanning, we process the chunks in such a way that each chunk is sorted, but the boundaries between chunks are not necessarily sorted. After doing so, we only need to merge them into one big, sorted sequence. How to efficiently merge sequences in-place will be explained later, in Section 4.3.3.

To transform each chunk into a sorted one, we choose a different method of sorting it, depending on how it got classified: We have to reverse the chunk if it was reversed, use the partition-friendly algorithm from Section 4.3.1, use the merge-friendly algorithm from Section 4.3.2, or sort it with the default algorithm — in our case IPS^{4}o. To save time in the merging phase, we join consecutive chunks that were classified as the same case: For example, if the 1st and 2nd chunk are sorted and the 3rd and 4th are not, we "ignore" the first half and sort the second half in one go. Thus, we will only have to merge two sequences instead of four if we had dealt with every chunk separately.

In order to detect the different cases, we calculate two metrics over our input:

`balance` We count how many pairs violate the condition `input[i] ⩽ input[i + 1]`.

`streaks` We count how many runs in the input are either increasing or strictly decreasing. This is done by "splitting" the input into slices of a fixed length (e.g., 32) and counting which of these slices are sorted or reversed.

This is everything we need to assign the right case to each chunk:

- if `balance` is zero, the chunk is sorted
- if `balance` is equal to the length of the chunk, it is reversed
- if `streaks` is bigger than 75% of all possible streaks, the chunk is merge-friendly

[1]In crumsort [15], there are four chunks, and they are names "quads".

- if `balance` is bigger than 75% or smaller than 25% of the chunk length, it is partition-friendly
- otherwise, the chunk is unsorted

Note that if a chunk is detected as both merge-friendly and partition-friendly, we will say that it is merge-friendly, because the `streaks` metric is more specific than the `balance` metric.

4.3.1. Partition-Friendly Algorithm

While comparing our algorithm to the other sorting algorithms from Chapter 2, we noticed that for patterns with few distinct values, IPS^{4}o lacks behind the quicksort variants. We suspect that this is the case because IPS^{4}o has to put more work into separating values into different buckets. When encountering only few distinct values (the extreme case being only two), this extra effort is not worth it because partitioning into two parts is just as effective but less work intensive. To use this finding to our advantage, we introduce the notion of partition-friendliness to detect the patterns where quicksort-based algorithms are faster and handle them accordingly.

We came up with two main ways of handling these partition-friendly inputs: We can switch to a quicksort-based algorithm, or we can use the same partitioning scheme as quicksort, but only for one iteration, and then switch back to IPS^{4}o. We argue that the second solution is more elegant, as it integrates better with our main algorithm and thus provides a potential performance gain over other algorithms (assuming IPS^{4}o is faster in general).

4.3.2. Merge-Friendly Algorithm

If the pattern detection classifies a chunk as merge-friendly, we know that the chunk contains many sorted subsequences. Sorting algorithms based on merge sort naturally use sorted subsequences. If implemented correctly, such a sorting algorithm can exploit this pattern and outperform other algorithms.

There are many options for choosing such an algorithm, but most of them are not in-place due to their merging and thus not suitable for us. Therefore, we mainly focus on three algorithms for our experiments: timsort, glidesort and quadsort. As already explained in Section 2.3, timsort requires auxiliary memory to work. While this is a dealbreaker for use in our final algorithm, it still provides a good baseline for evaluation of the other two algorithms. Although glidesort and quadsort are both merge based algorithms, they both provide an interface to pass in a buffer (of constant size) which they will then use exclusively as auxiliary storage, thus making them in-place. When using these algorithms for sorting a chunk that was classified as merge-friendly, our measurements indicate glidesort to be faster than quadsort and even faster than timsort in many cases.

4.3.3. Fast In-Place Merge

From the previous phase, we know that every chunk is now sorted. It might even be that the whole sequence is already sorted, but that is not guaranteed. For example, when encountering the unsorted tail pattern, all but one chunk will be a single sorted sequence, with only the last chunk being unsorted. In its original implementation, the `crum_analyze`-function from crumsort [15] (see Section 2.5) merges the sequences with the in-place merge algorithm called "blitmerge" by the same author. As blitmerge was also implemented in crumsort's Rust port [17], it was very straightforward for us to integrate into our version of `crum_analyze`. We also integrated the in-place merger implemented in glidesort, which outperforms blitmerge, according to our measurements.

Glidemerge: Glidemerge is the merge algorithm used in glidesort [18] (see Section 2.6). Its most important function `double_merge` takes two contiguous, sorted sequences and merges them. We use binary search to find the biggest n for which the following holds: When we swap the last n elements of the left sequence with the first n elements of the right sequence, we can merge both resulting sequences independently of each other and obtain a fully sorted sequence. We will name the resulting parts (l, a, b, r), where a and b have length n, and l and r are the remaining parts.

If a is bigger than our buffer size, we swap a and b, so the resulting sequence looks like this: (l, b, a, r). We can now recurse into merging l with b and a with r, respectively. We repeat this process until the parts are small enough to fit into the buffer.

If a is small enough, we copy it into our buffer. We are now left with a gap between l and b. Since a and b are the same size, the gap is big enough to merge l with b. After doing that, we now have a gap where b was before and can use that to merge a with r.

4.3.4. Parallel

We tried two different approaches to parallelize our adaptiveness:

Approach A: Analyze before Sorting: This approach tries to parallelize the detection step. Each thread gets an equal part of the input (equivalent to a stripe in IPS^{4}o) and checks if it is sorted or reversed. A stripe corresponds to a chunk in the sequential case. The parallel version of IPS^{4}o is more efficient on bigger inputs. That is why we omit classifying the stripes as merge-friendly and partition-friendly — to leave the consecutive stripes that were classified to be the same as big as possible. The problem with this approach is that it requires merging the stripes at the end. To our knowledge, there currently is no in-place parallel merger. While we can parallelize the recursive calls of our sequential in-place merger, this will not give us a very good speedup. Much of the work is done in the first few recursions, where only very few threads are used.

Approach B: Analyze on Switch to Sequential: The second approach is very straightforward: Like the non-adaptive IPS^{4}o (see Section 2.2.5), we start by partitioning the input in parallel. Only when we switch from the parallel sorting algorithm to the sequential one, we start looking for patterns by calling the (sequential) `crum_analyze`-function. With this, we avoid the problem of having to merge in parallel. For this approach to work well, the algorithm "hopes" that the first IPS^{4}o recursion leaves existing patterns nearly intact, so that they can be found and exploited after changing to the sequential code.

5. Results

In this chapter, we discuss the performance impact of our different optimizations regarding the adaptiveness of IPS^{4}o. We implemented our adaptive approaches in Rust, so in order to give a fair comparison, we explicitly don't compare the adaptive implementation with the C++ implementation, but only with the Rust one. For a comparison between Rust and C++, see Chapter 3. For a comparison between IPS^{4}o and other sorting algorithms, refer to the original paper [2].

5.1. Experimental Setup

We measured our algorithm using `Criterion.rs`[1]. The Rust code was compiled using the `nightly-2023-01-30` toolchain. For the measurements presented in this chapter, we used an *AMD Ryzen 9 3950X*, a 16-core CPU with 64GB of RAM. Further measurements, performed on an *Intel i7 11700*, a 8-core CPU with 64GB of RAM, can be found in Appendix A.

We ran our benchmarks with a number of different patterns, most of them were taken from `sort-research-rs` [25]:

- The values in the input follow a uniform distribution. (*uniform*)
- All elements are in increasing order. (*sorted*)
- All elements are in decreasing order. (*reversed*)
- The input follows a saw pattern[2] with $\log_2(n)$ teeth. Whether a given tooth is increasing or decreasing, is chosen randomly. (*saw-long*)
- The first $(1 - x\%)$ of elements are sorted, the remaining $x\%$ of elements are uniformly distributed. (*unsorted-tail-x%*)
- $x\%$ are sorted, the rest are random elements at random indices. (*sorted-x%*)
- $x\%$ are zeroes, the rest are random elements at random indices. (*zeroes-x%*)

Our plots show the mean running time over 20 samples across a range of input sizes, together with the upper and lower bound of the 95% confidence interval. Because the confidence interval is pretty narrow most of the time, the corresponding markers are often hidden behind the main marker.

5.2. Equidistant Sampling

This section discusses the results of implementing equidistant sampling instead of random sampling in sequential IPS^{4}o, as described in Section 4.2. Figure 5 shows the running times of the *uniform* and the *sorted-99%* pattern.

[1]https://github.com/bheisler/criterion.rs

[2]A saw pattern is a concatenation of sorted sequences (teeth).

We see that for the *sorted-99%* pattern, equidistant sampling performs a lot better than random sampling, being 1.3 times faster. This is the result that we expected for this measurement, because it presents the best-case scenario for equidistant sampling: The quality of the splitters is the same or better than with random sampling, and the sample from which the splitters are chosen is nearly sorted already.

Even for the *uniform* pattern, equidistant sampling performs better than random sampling, but only by a factor of 1.03. We suspect that the main reason behind the performance improvement is the predictable access pattern which the CPU can use to apply prefetching of values.

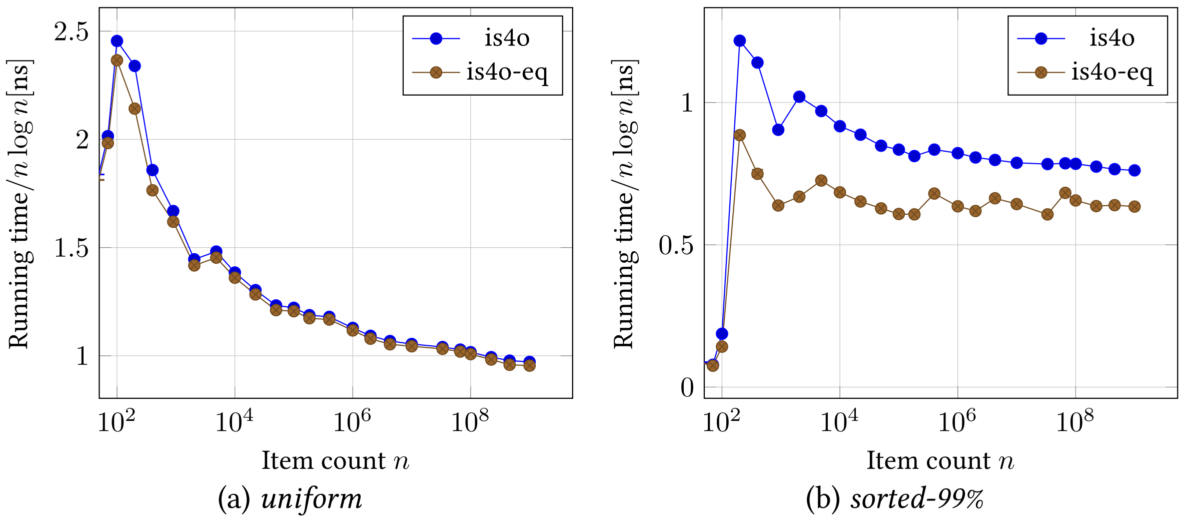


Figure 5: Random vs. equidistant sampling, performed on an *AMD Ryzen 9 3950X*.

5.3. Preliminary Pattern Detection

Figure 6 presents the results of using preliminary pattern search (in addition to equidistant sampling) to increase the adaptiveness of sequential IPS^{4}o, as described in Section 4.3.

A very important property for any adaptive sorting algorithm is to ensure that it still performs well on random data, because that is probably the most common pattern. In Figure 6a, we can see that this property holds true, as both the adaptive and the non-adaptive version of IPS^{4}o perform equally well. Note that we still take a small performance hit for random data when using the `crum_analyze` function. But as we have seen in Section 5.2, equidistant sampling gives us an edge over random sampling — this edge is now lost. The benefit is that our algorithm performs a lot better for other patters, as we will see in the following.

The unsorted tail pattern is an easy pattern to detect and the payoff for doing so is very high, as we see in Figure 6b. For sizes upwards of 10 000 elements, our algorithm is faster than normal IPS^{4}o by a factor of 6.67. This is about what we expect, because our `crum_analyze` function is configured to split the input into eight equal parts (chunks). The first seven chunks are detected as sorted, so we only have to sort the last eighth of the input and merge it with the first seven chunks.

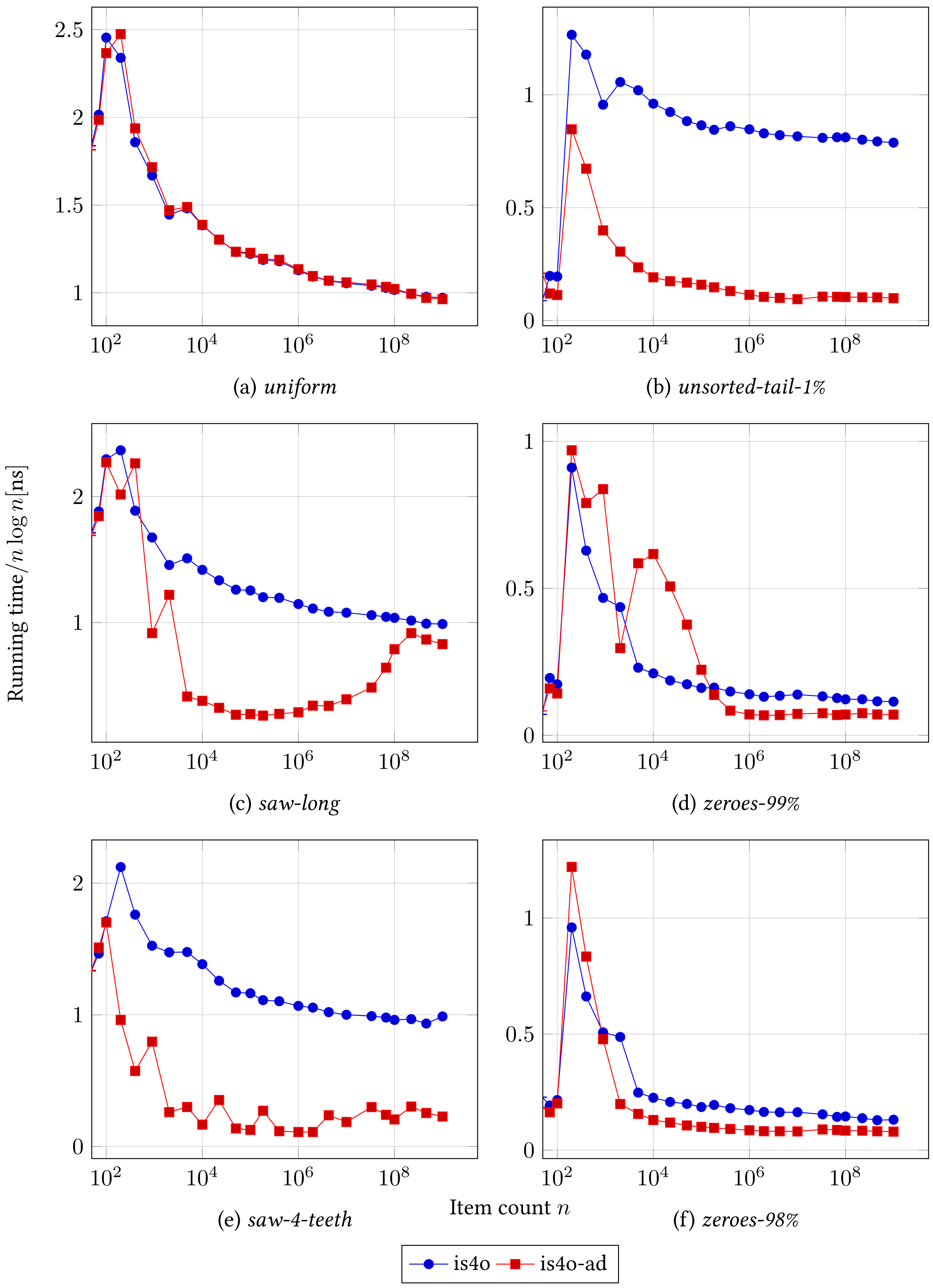


Figure 6: Standard vs. adaptive version of IS⁴o across different patterns.

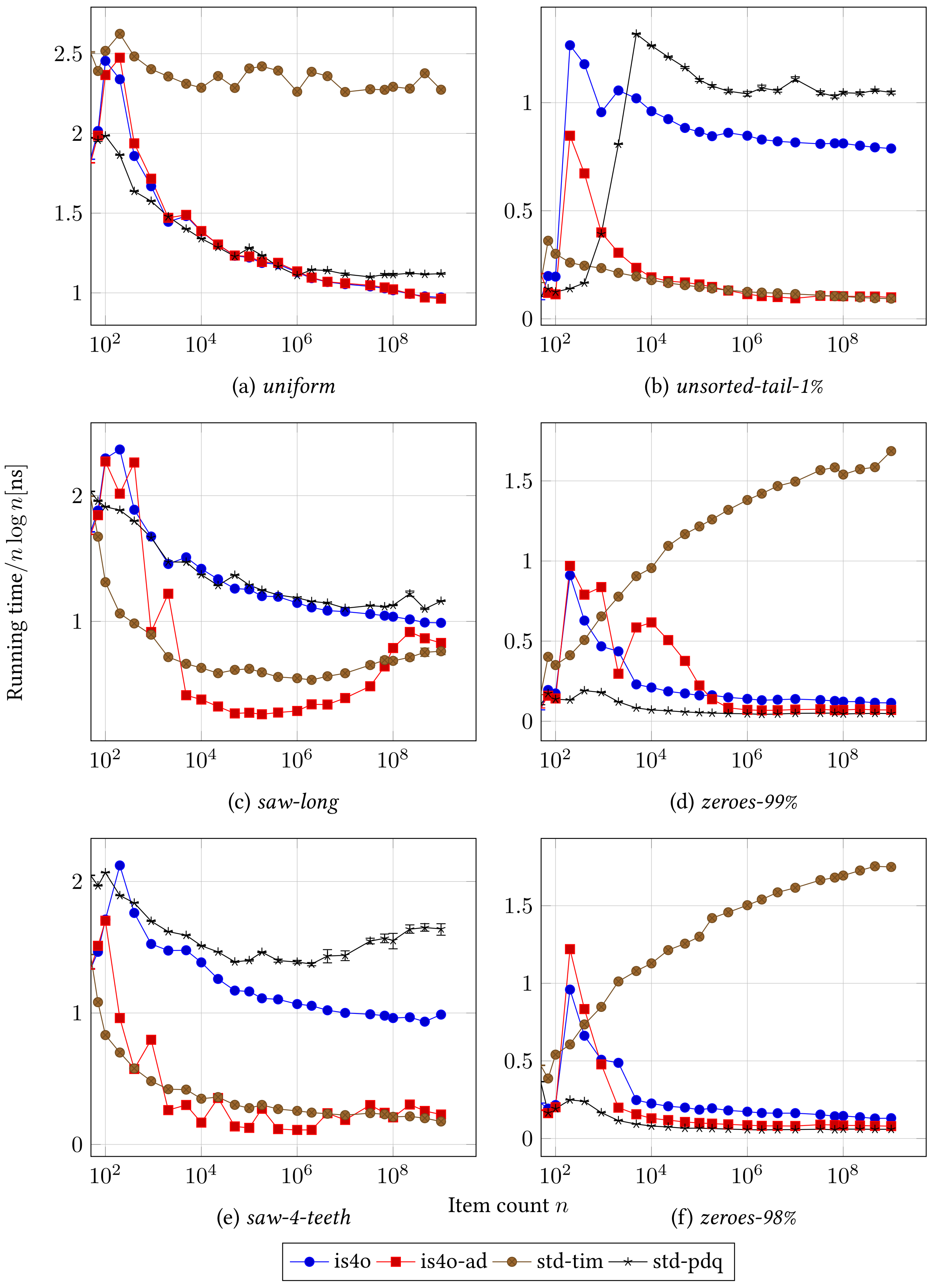


Figure 7: Standard and adaptive version of IS⁴o, together with both sorting algorithms of the Rust standard library. The stable algorithm is based on timsort, the unstable algorithm is based on pattern-defeating quicksort.

Because IPS^{4}o is based on partitioning the input to sort it, the algorithm does not make much (if any) use of saw-like patterns. Our approach avoids this issue by detecting such a pattern and changing to an algorithm that is based on mergesort. In Figure 6c, we see that this approach outperforms IPS^{4}o by a factor of 3.81 for n between 10^4 and 10^7. We can also see that our detection heuristic doesn't work perfectly: The outlier at $n = 2048$ is caused by misclassifying the input as not merge-friendly. For $n \geq 10^7$, the detection classifies everything correctly, but the in-place mergesort algorithm we use (glidesort) slows down. We also included the easier *saw-4-teeth*-pattern: Here we can see that the merge sort algorithm performs more consistently across the different input sizes, making it around 5 times faster than the non-adaptive version.

In Figure 6d, we see that on the pattern *zeroes-99%*, the adaptive version on average is 1.79 times faster for inputs with $n \geq 4 \cdot 10^5$, by classifying them as partition-friendly. For inputs smaller than that, the algorithm misclassifies part of the input as merge-friendly, which takes priority over partition-friendliness. As we can see in the plot, the merge-friendly algorithm performs worse than IPS^{4}o on this particular pattern. This results in the adaptive version being slower by a factor of 0.75 on inputs with $n < 4 \cdot 10^5$. If we instead look at the pattern *zeroes-98%* (Figure 6f), the pattern-detection classifies the input correctly as partition-friendly and is thus on average 1.75 times faster for inputs starting at $n \geq 900$.

To provide a clearer way of comparing the two approaches, we only plot `is4o` and `is4o-ad` in Figure 6. In Figure 7, we compare our adaptive algorithm with the two standard library implementations, `sort` and `sort_unstable` [14]. They are based on timsort and pattern-defeating quicksort, respectively. We see that on each of the presented patterns, one of the standard library algorithms performs significantly better than non-adaptive IPS^{4}o and the other std-sort. Through our adaptive approach, we managed to make IPS^{4}o very efficient across *all* patterns, both on patterns where `sort` is better and on patterns where `sort_unstable` is better.

Because the original IPS^{4}o implementation already performs a primitive preliminary pattern search by checking if the input is sorted increasingly or decreasingly, we did not get any improvements on these two patterns.

5.4. Parallel

In Figure 8, we see the adaptive parallel approach described in Section 4.3.4B. Note that we are only interested in inputs with size $n \geq 10^5$, because for smaller inputs the sequential algorithm is used, which we already discussed in Chapter 4.

As expected, the performance for the *uniform* pattern is nearly the same for both algorithms. For the *unsorted-tail-1%* pattern, the adaptive approach is faster by a factor of 1.75, for inputs of the size n between 10^5 and $5 \cdot 10^8$. For our biggest input, $n = 10^9$, it is 0.89 times slower. The *saw-long* pattern shows similar results: For the "smaller" sizes up to $n \leq 5 \cdot 10^8$ elements, the adaptive algorithm is faster by a factor of 1.27, but for $n = 10^9$ it is significantly slower by a factor of 0.57. The slowdown for bigger inputs happens because the algorithms detects the chunks as merge-friendly, despite IPS^{4}o being more efficient than our merge-friendly sorting algorithm. Different from the sequential algorithm, we do not get any benefit from detecting the *zeroes-99%* or the *zeroes-98%* patterns.

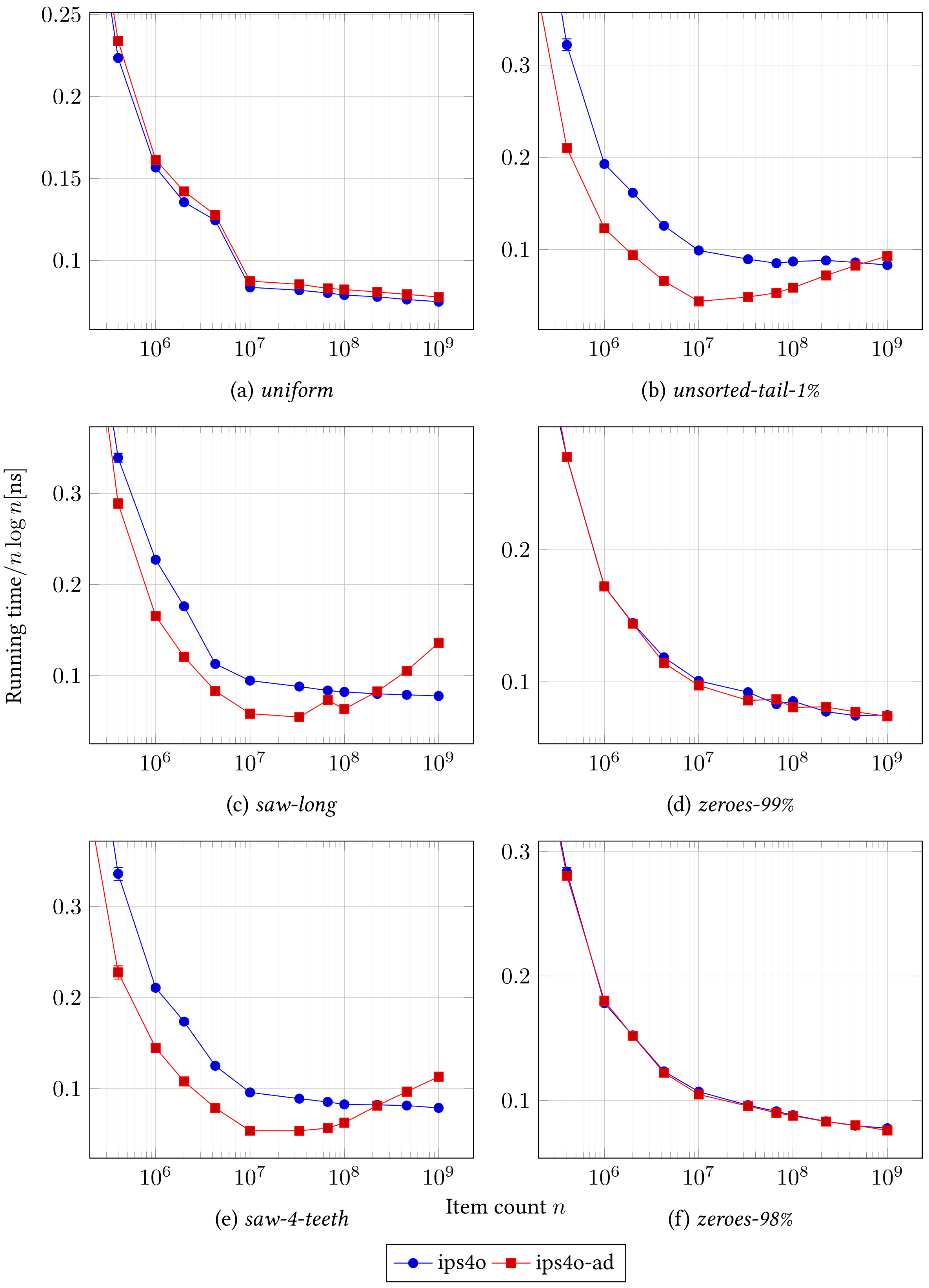


Figure 8: Standard vs. adaptive version of parallel IPS⁴o as described in Section 4.3.4B

6. Conclusion and Future Work

We successfully adapted sequential IPS^{4}o to significantly accelerate the sorting of various input patterns. Although we didn't manage to include a good solution *into* IPS^{4}o, the approach of finding nearly sorted patterns *before* starting to sort with IPS^{4}o proved to be effective. Our approach works by splitting the input into multiple parts of equal length (chunks) to scan them and potentially change to another sorting algorithm. For *unsorted-tail-1%*, the input pattern with the biggest improvement, our approach is over 6 times faster than non-adaptive IPS^{4}o. We also used this approach in our parallel version of IPS^{4}o. There, the approach only worked well for inputs smaller than 10^8 — for bigger inputs, our approach is slower than the non-adaptive implementation. Additionally, we were able to speed up IPS^{4}o by changing its splitter selection strategy from random sampling to equidistant sampling.

Furthermore, we reimplemented IPS^{4}o in Rust, providing a case study into the performance of Rust compared to C++. As seen in Section 3.2, we needed to use *unsafe* code to be competitive with the original implementation. Further research is needed to find why our Rust implementation still performs worse than the C++ one. To make our implementation "production-ready", it should be possible to use it as a fill-in replacement for Rust's standard library sorting algorithms. For this to be the case, the implementation has to be changed to accept any generic type without requiring the `Clone` and `Default` traits and should fulfill the properties described in Section 3.3.

It would be interesting to see our approach integrated into other sorting algorithms, for example those found in the Rust standard library or the C++ implementation of IPS^{4}o. An empirical study to find out what the most common patterns are in the real world would help to further improve the preliminary pattern detection.

Bibliography

[1] C. E. Leiserson, R. L. Rivest, T. H. Cormen, and C. Stein, *Introduction to Algorithms*, vol. 3, MIT press Cambridge, MA, USA, 1994.

[2] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place Parallel Super Scalar Samplesort (IPSSSSo)", *Arxiv Preprint Arxiv:1705.02257*, 2017. [Online]. Available: https://arxiv.org/abs/1705.02257

[3] T. Peters, "Timsort". https://svn.python.org/projects/python/trunk/Objects/listsort.txt (accessed: Sep. 5, 2023).

[4] O. R. L. Peters, "Pattern-defeating Quicksort", *Corr*, 2021. [Online]. Available: https://arxiv.org/abs/2106.05123

[5] T. Heinen, "ips4o_rs", 2023. [Online]. Available: https://github.com/Heinenen/ips4o_rs

[6] C. A. Hoare, "Quicksort", *Comput. J.*, vol. 5, no. 1, pp. 10–16, 1962.

[7] P. Tsigas, and Y. Zhang, "A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000", in *Eleventh Euromicro Conf. Parallel, Distrib. Network-Based Processing, 2003. Proceedings.*, 2003, pp. 372–381.

[8] J. Singler, P. Sanders, and F. Putze, "MCSTL: The multi-core standard template library", in *Euro-Par 2007 Parallel Processing: 13th Int. Euro-Par Conference, Rennes, France, August 28-31, 2007. Proc. 13*, 2007, pp. 682–694.

[9] S. Edelkamp, and A. Weiß, "Blockquicksort: Avoiding branch mispredictions in quicksort", *J. Exp. Algorithmics (JEA)*, vol. 24, pp. 1–22, 2019.

[10] W. D. Frazer, and A. C. McKellar, "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting", *J. Acm*, vol. 17, no. 3, p. 496, Jul. 1970. [Online]. Available: https://doi.org/10.1145/321592.321600

[11] P. Sanders, and S. Winkel, "Super scalar sample sort", in *Eur. Symp. Algorithms*, 2004, pp. 784–796.

[12] "Java™ Platform, Standard Edition 8 API Specification". https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html (accessed: Sep. 5, 2023).

[13] "Implementation of Timsort in the Swift standard libary". https://github.com/apple/swift/blob/main/stdlib/public/core/Sort.swift (accessed: Sep. 5, 2023).

[14] "Rust Language Documentation: Primitive Type slice". https://doc.rust-lang.org/std/primitive.slice.html (accessed: Sep. 5, 2023).

[15] I. van den Hoven, “crumsort”, 2023. Accessed: Sep. 5, 2023. [Online]. Available: https://github.com/scandum/crumsort

[16] I. van den Hoven, “quadsort”. Accessed: Sep. 24, 2023. [Online]. Available: https://github.com/scandum/quadsort

[17] “crumsort-rs”, 2022. Accessed: Sep. 6, 2023. [Online]. Available: https://github.com/google/crumsort-rs

[18] O. R. L. Peters, “glidesort”, 2023. Accessed: Sep. 18, 2023. [Online]. Available: https://github.com/orlp/glidesort

[19] J. I. Munro, and S. Wild, “Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs”, in *26th Annu. Eur. Symp. Algorithms (ESA 2018)* in Leibniz International Proceedings in Informatics (Lipics), vol. 112, Dagstuhl, Germany, 2018, pp. 1–16, doi: 10.4230/LIPIcs.ESA.2018.63.

[20] I. van den Hoven, “fluxsort”. Accessed: Sep. 24, 2023. [Online]. Available: https://github.com/scandum/fluxsort

[21] “The Rust Programming Language (old website)”. https://prev.rust-lang.org/id-ID/faq.html (accessed: Sep. 1, 2023).

[22] G. Thomas, “A proactive approach to more secure code”. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (accessed: Sep. 1, 2023).

[23] N. D. Matsakis, and F. S. Klock, “The Rust Language”, *ACM Sigada Ada Lett.*, vol. 34, no. 3, pp. 103–104, 2014.

[24] V. Estivill-Castro, and D. Wood, “A survey of adaptive sorting algorithms”, *ACM Comput. Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.

[25] “sort-research-rs”. Accessed: Sep. 23, 2023. [Online]. Available: https://github.com/Voultapher/sort-research-rs

A. Appendix

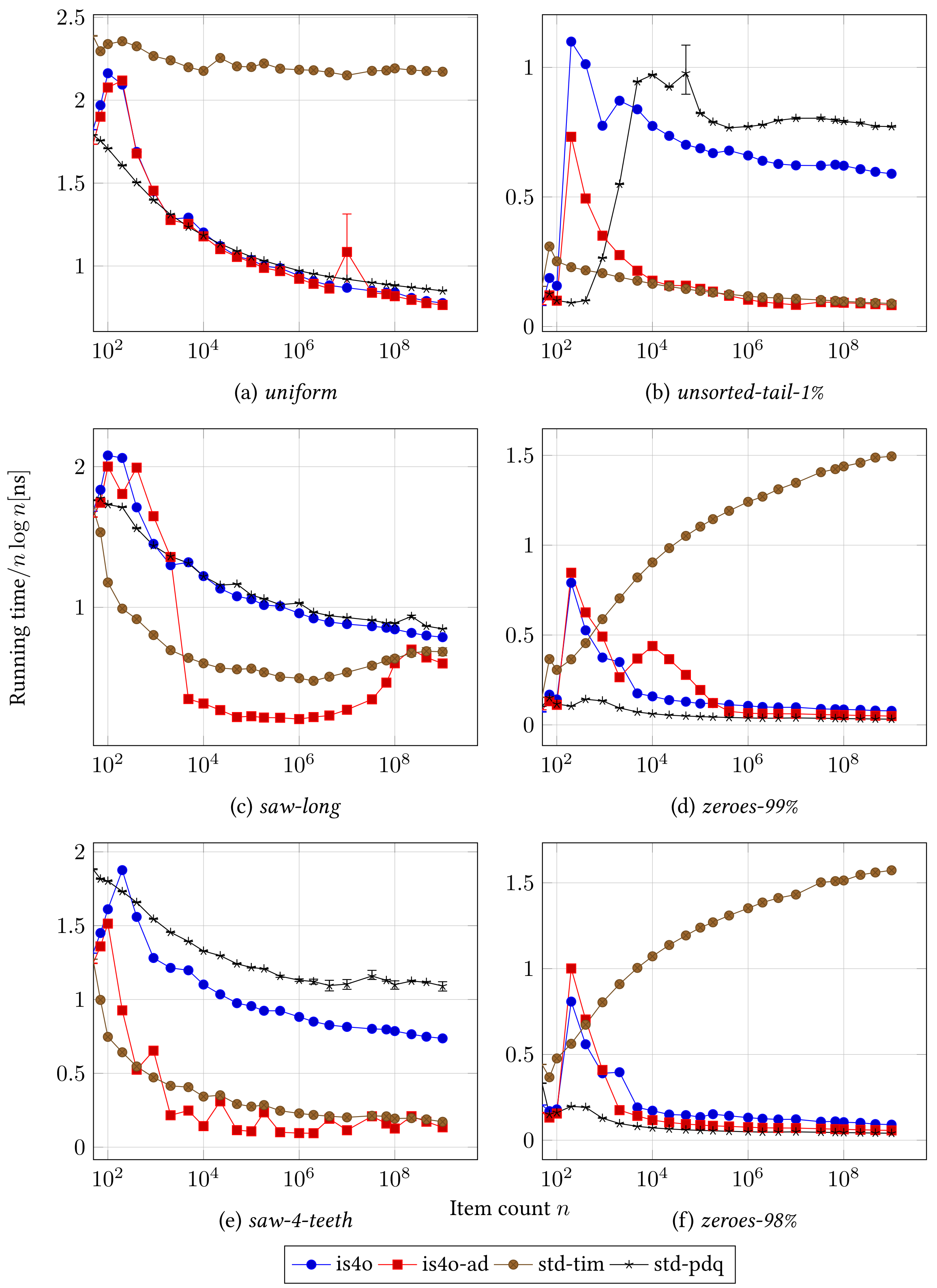


Figure 9: Standard and adaptive version of IS⁴o, together with both sorting algorithms of the Rust standard library. Measured on an *Intel i7 11700* 8-core CPU.

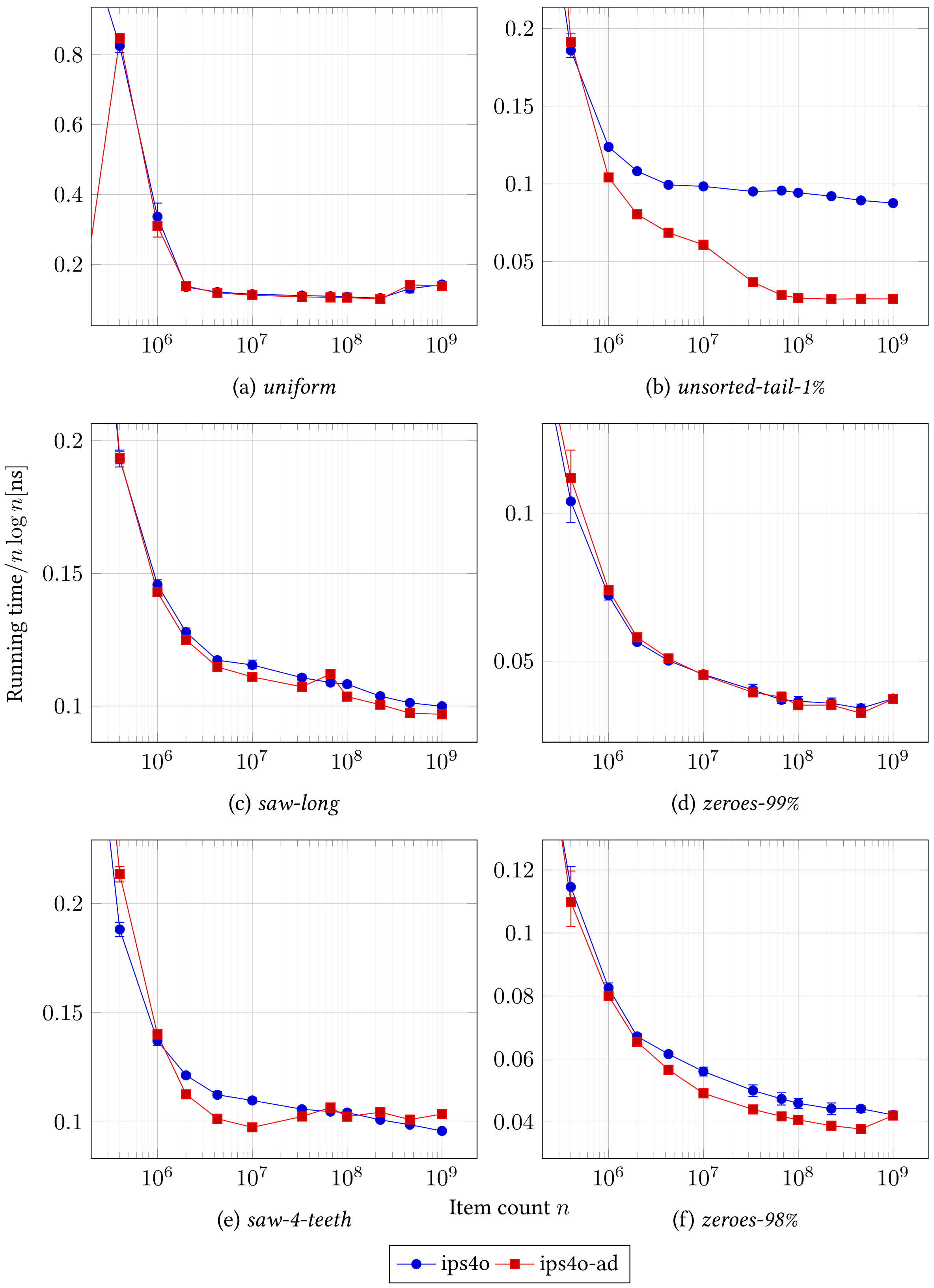


Figure 10: Standard vs. adaptive version of parallel IPS⁴o as described in Section 4.3.4B. Measured on an *Intel i7 11700* 8-core CPU.