Glidesort

Efficient In-Memory Adaptive Stable Sorting on Modern Hardware

Orson Peters
Research done at CWI Database Architectures group
What is glidesort?

- General purpose **stable comparison sort**.
- A hybrid of mergesort, quicksort and block insertion sort.
- **Robustly** adaptive to pre-sorted and low-cardinality inputs.
- Reference implementation in (unsafe) Rust.

Drop-in for `[T]::sort`
Stable quicksort?

Yes!

https://github.com/scandum/fluxsort

Igor van den Hoven
Quicksort is an in-place sorting algorithm. Developed by British computer scientist Tony Hoare in 1959[1] and published in 1961,[2] it is still a commonly used algorithm for sorting. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.[3][contradictory]

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort.[4] The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.
std::stable_sort uses O(n) auxiliary memory and no one bats an eye.

Stable quicksort uses O(n) auxiliary memory and everyone loses their minds.
Adaptive sorting

adapt verb
To change your behaviour in order to deal more successfully with a new situation.
Divide and conquer

**Merge**
- Mergesort
- Timsort
- Powersort

**Partition**
- Quicksort
- Samplesort
- Radix sort

Fundamentally **bottom-up**.
Can be *adaptive* to pre-sorted runs.

Fundamentally **top-down**.
Can be *adaptive* to low-cardinality inputs.
Low-cardinality inputs

SELECT * FROM customers ORDER BY city;

SELECT * FROM cars ORDER BY brand;
Adaptive quicksort

Idea:

- During partitioning detect buckets of all-equal elements.

Challenges:

- Minimize overhead comparisons.
- Avoiding three-way comparisons.

Rough history:

- Quicksort (Hoare, 1961)
- Dutch national flag problem (Dijkstra, 1976)
- Unix qsort (Bentley-McIlroy, 1992)
- Pattern-defeating quicksort (Orson Peters, 2015)
Adaptive pdqsort

partition-left

\[
\begin{array}{c c c c}
\leq P & P & > P \\
\end{array}
\]

partition-right

\[
\begin{array}{c c c c}
< P & P & \geq P \\
\end{array}
\]

See earlier talk on pdqsort:  
https://www.youtube.com/watch?v=jz-PBiWwNjc
Adaptive pdqsort

Each value can be a pivot at most twice:

On average $O(n \log(k))$ for $k$ distinct values
Adaptive mergesort

Idea:

- Merge pre-existing runs.

Challenges:

- Minimize unbalanced merges.
- Storing run information.

Rough history:

- Mergesort (von Neumann, 1945)
- Natural mergesort (Knuth, 1973)
- Timsort (Tim Peters, 2002)
- Powersort (Munro-Wild, 2018)
Powersort

Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs

J. Ian Munro, Sebastian Wild

Outline of main loop:

1. run = create_run(prev_run.end, array)
2. p = power(prev_run, run)
3. while peek(stack).p > p:
   run = merge(pop(stack).run, run)
4. push(stack, (run, p))
5. prev_run = run

- Stack is at most \( \log_2(n) \) runs.
- Provably creates good and stable merge sequences heuristically.
- \texttt{create\_run} can take advantage of existing runs in input.
A problem emerges

<table>
<thead>
<tr>
<th>Merge</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Mergesort</td>
<td>● Quicksort</td>
</tr>
<tr>
<td>● Timsort</td>
<td>● Samplesort</td>
</tr>
<tr>
<td>● Powersort</td>
<td>● Radix sort</td>
</tr>
</tbody>
</table>

Fundamentally **bottom-up.** Can be *adaptive* to pre-sorted runs.

Fundamentally **top-down.** Can be *adaptive* to low-cardinality inputs.
Glide.
A soaring bird only flaps its wings when necessary.

```java
enum Run {
    Unsorted(Range),
    Sorted(Range),
    Concatenated((Range, Range)),
}
```
Powersort

Outline of main loop:

1. run = create_run(prev_run.end, array)
2. p = power(prev_run, run)
3. while peek(stack).p > p:
   run = merge(pop(stack).run, run)
4. push(stack, (run, p))
5. prev_run = run

- Stack is at most \( \log_2(n) \) runs.
- Provably creates good and stable merge sequences.
- create_run can take advantage of existing runs in input.
Creating a ‘run’

create_run(start, array)

1. Scan array while nondecreasing (or strictly descending) to find run.
2. If bigger than RUN_THRESHOLD, (reverse) and return Run::Sorted(run).
3. Otherwise, return a Run::Unsorted with length RUN_THRESHOLD.
fn logical_merge(left_run: Run, right_run: Run) -> Run {
    match (left_run, right_run) {
        (Unsorted(l), Unsorted(r)) => {
            if l.len() + r.len() <= scratch_space.len() {
                Unsorted(Range(l.begin, r.end))
            } else {
                Concatenated(quicksort(l), quicksort(r))
            }
        },
        (Sorted(l), Sorted(r)) => Concatenated((l, r)),
        (Unsorted(l), right_run) => logical_merge(quicksort(l), right_run),
        (left_run, Unsorted(r)) => logical_merge(left_run, quicksort(r)),
        (Concatenated(l), Sorted(r)) => physical_triple_merge(l.0, l.1, r),
        (Sorted(l), Concatenated(r)) => physical_triple_merge(l, r.0, r.1),
        (Concatenated(l), Concatenated(r)) => physical_quad_merge(l.0, l.1, r.0, r.1),
    }
}
fn logical_merge(left_run: Run, right_run: Run) -> Run {
    match (left_run, right_run) {
        (Unsorted(l), Unsorted(r)) => {
            if l.len() + r.len() <= scratch_space.len() {
                Unsorted(Range(l.begin, r.end))
            } else {
                Concatenated(quicksort(l), quicksort(r))
            }
        },
        (Sorted(l), Sorted(r)) => Concatenated((l, r)),
        (Unsorted(l), right_run) => logical_merge(quicksort(l), right_run),
        (left_run, Unsorted(r)) => logical_merge(left_run, quicksort(r)),
        (Concatenated(l), Sorted(r)) => physical_triple_merge(l.0, l.1, r),
        (Sorted(l), Concatenated(r)) => physical_triple_merge(l, r.0, r.1),
        (Concatenated(l), Concatenated(r)) => physical_quad_merge(l.0, l.1, r.0, r.1),
    }
}
fn logical_merge(left_run: Run, right_run: Run) -> Run {
    match (left_run, right_run) {
        (Unsorted(l), Unsorted(r)) => {
            if l.len() + r.len() <= scratch_space.len() {
                Unsorted(Range(l.begin, r.end))
            } else {
                Concatenated(quicksort(l), quicksort(r))
            }
        },

        (Sorted(l), Sorted(r)) => Concatenated((l, r)),
        (Unsorted(l), right_run) => logical_merge(quicksort(l), right_run),
        (left_run, Unsorted(r)) => logical_merge(left_run, quicksort(r)),
        (Concatenated(l), Sorted(r)) => physical_triple_merge(l.0, l.1, r),
        (Sorted(l), Concatenated(r)) => physical_triple_merge(l, r.0, r.1),
        (Concatenated(l), Concatenated(r)) => physical_quad_merge(l.0, l.1, r.0, r.1),
    }
}
fn logical_merge(left_run: Run, right_run: Run) -> Run {
  match (left_run, right_run) {
    (Unsorted(l), Unsorted(r)) => {
      if l.len() + r.len() <= scratch_space.len() {
        Unsorted(Range(l.begin, r.end))
      } else {
        Concatenated(quicksort(l), quicksort(r))
      }
    },
    (Sorted(l), Sorted(r)) => Concatenated((l, r)),
    (Unsorted(l), right_run) => logical_merge(quicksort(l), right_run),
    (left_run, Unsorted(r)) => logical_merge(left_run, quicksort(r)),
    (Concatenated(l), Sorted(r)) => physical_triple_merge(l.0, l.1, r),
    (Sorted(l), Concatenated(r)) => physical_triple_merge(l, r.0, r.1),
    (Concatenated(l), Concatenated(r)) => physical_quad_merge(l.0, l.1, r.0, r.1),
  }
}
Merging two runs

```rust
fn logical_merge(left_run: Run, right_run: Run) -> Run {
    match (left_run, right_run) {
        (Unsorted(l), Unsorted(r)) => {
            if l.len() + r.len() <= scratch_space.len() {
                Unsorted(Range(l.begin, r.end))
            } else {
                Concatenated(quicksort(l), quicksort(r))
            }
        },
        (Sorted(l), Sorted(r)) => Concatenated((l, r)),
        (Unsorted(l), right_run) => logical_merge(quicksort(l), right_run),
        (left_run, Unsorted(r)) => logical_merge(left_run, quicksort(r)),
        (Concatenated(l), Sorted(r)) => physical_triple_merge(l.0, l.1, r),
        (Sorted(l), Concatenated(r)) => physical_triple_merge(l, r.0, r.1),
        (Concatenated(l), Concatenated(r)) => physical_quad_merge(l.0, l.1, r.0, r.1),
    }
}
```
Glidesort main loop summary

- Extension of powersort (but applicable to any natural stable mergesort)
- Does not eagerly sort small runs
- Defers physically merging as long as possible
- Transforms sorting problem into quicksorts and triple/quad merges
- Adaptive to pre-sorted runs and low-cardinality inputs
Why triple/quad merges?

1. Ping-pong merging
2. Bidirectional merging
3. Parallel merging
Ping-pong merges

Patience is a Virtue: Revisiting Merge and Sort on Modern Processors
Badrish Chandramouli, Jonathan Goldstein

https://github.com/scandum/quadsort
Igor van den Hoven

- Traditional merge memcpy’s smaller array to scratch space before merging back
- Triple/quad merges can merge both into the scratch space and back
Bidirectional merging

- Destination and source arrays disjoint?
- Merge from both ends!

```rust
loop {
    let common = self.left.len().min(self.right.len());
    if common == 0 { break; }
    for _ in 0..common / 4 {
        self.branchless_merge_one_at_begin(is_less);
        self.branchless_merge_one_at_end(is_less);
        self.branchless_merge_one_at_begin(is_less);
        self.branchless_merge_one_at_end(is_less);
    }
    for _ in 0..common % 4 {
        self.branchless_merge_one_at_begin(is_less);
    }
}

'Parity merge'
https://github.com/scandum/quadsort
Igor van den Hoven
Modern processors are:

1. superscalar
2. out-of-order
3. deeply pipelined
Branchless merge (at begin)

Branch Mispredictions Don't Affect Mergesort
Amr Elmasry, Jyrki Katajainen & Max Stenmark

```rust
let left_scan = self.left.begin();
let right_scan = self.right.begin();
let right_less = is_less(&*right_scan, &*left_scan);
let src = select(right_less, right_scan, left_scan);
ptr::copy_nonoverlapping(src, self.out.begin(), 1);
self.out.add_begin(1);
self.right.add_begin(right_less as usize);
self.left.add_begin(!(right_less) as usize);
```
Dependencies

```rust
let left_scan = self.left.begin();
let right_scan = self.right.begin();
let right_less = is_less(*right_scan, *left_scan);
let src = select(right_less, right_scan, left_scan);
ptr::copy_nonoverlapping(src, self.out.begin(), 1);
self.out.add_begin(1);
selh.right.add_begin(right_less as usize);
selh.left.add_begin(!right_less as usize);
```
Interleave independent branchless loops.

(within reason - don't spill to stack / overload prefetcher)
‘Parallel’ merging

- First step in quad merge has two independent merges
- Can parallelize, but no threads…
- …interleave loops
Creating more parallelism

- With binary search we can find crossover point
  \[ L[\text{len}(L)-k] > R[k] \]
- Swap last/first \( k \) elements
- Out-of-place merge? Swap is free!
- \( O(n \log(n)^2) \) fallback for stable merging with \( O(1) \) buffer
Bidirectional stable partitioning

Same principle, interleaving two independent scans
Bidirectional stable partitioning

- Recursion is a bit more involved...
- Input partially in scratch, partially in destination
- Invariant $|\text{dest}| = |\text{scratch}| = |L| + |R|$
Experimental setup

Your mileage may vary

macOS Monterey 12.1

Rust nightly, release profile
Link-time optimization = “thin”

Clang 13.0

g++  -std=c++17  -O2

2021 Apple M1 Pro high perf. core
Max frequency:  3.2GHz
L1:   128 KiB (data), 196 KiB (instr)
L2:   12 MiB (shared)
L3:   n/a

All figures reported are medians, all experiments single-threaded.
<table>
<thead>
<tr>
<th>ns/N log₂ N</th>
<th>N = 2²⁴ (5x cache)</th>
<th>32-bit integers</th>
<th>a &lt; b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stable</td>
<td>Buffer</td>
<td>Shuffled</td>
</tr>
<tr>
<td>glidesort</td>
<td>Yes</td>
<td>n/2</td>
<td>0.624</td>
</tr>
<tr>
<td>glidesort1024</td>
<td>Yes</td>
<td>1024</td>
<td>1.373</td>
</tr>
<tr>
<td>Rust stable</td>
<td>Yes</td>
<td>n/2</td>
<td>2.710</td>
</tr>
<tr>
<td>std::stable_sort</td>
<td>Yes</td>
<td>n/2</td>
<td>3.012</td>
</tr>
<tr>
<td>cpp-timsort</td>
<td>Yes</td>
<td>n/2</td>
<td>3.579</td>
</tr>
<tr>
<td>pdqsort</td>
<td>No</td>
<td>O(1)</td>
<td>0.912</td>
</tr>
<tr>
<td>Rust unstable</td>
<td>No</td>
<td>O(1)</td>
<td>1.136</td>
</tr>
<tr>
<td>std::sort</td>
<td>No</td>
<td>O(1)</td>
<td>2.629</td>
</tr>
</tbody>
</table>

~4.3x faster than Rust stable sort
~4.7x faster than std::stable_sort for random data

Apple M1 MacBook Pro
Stable sort observable, cardinality 256

<table>
<thead>
<tr>
<th>ns/N log₂ N</th>
<th>N = 2²⁴ (5x cache)</th>
<th>32-bit integers</th>
<th>a % 256 &lt; b % 256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stable</td>
<td>Buffer</td>
<td>Shuffled</td>
</tr>
<tr>
<td>glidesort</td>
<td>Yes</td>
<td>n/2</td>
<td>0.261</td>
</tr>
<tr>
<td>glidesort1024</td>
<td>Yes</td>
<td>1024</td>
<td>1.212</td>
</tr>
<tr>
<td>Rust stable</td>
<td>Yes</td>
<td>n/2</td>
<td>3.334</td>
</tr>
<tr>
<td>std::stable_sort</td>
<td>Yes</td>
<td>n/2</td>
<td>1.714</td>
</tr>
<tr>
<td>cpp-timsort</td>
<td>Yes</td>
<td>n/2</td>
<td>2.102</td>
</tr>
<tr>
<td>pdqsort</td>
<td>No</td>
<td>O(1)</td>
<td>1.043</td>
</tr>
<tr>
<td>Rust unstable</td>
<td>No</td>
<td>O(1)</td>
<td>0.348</td>
</tr>
<tr>
<td>std::sort</td>
<td>No</td>
<td>O(1)</td>
<td>1.048</td>
</tr>
</tbody>
</table>

~12.8x faster than Rust stable sort
~6.6x faster than std::stable_sort for random data

Apple M1 Macbook Pro
Released now!

github.com/orlp/glidesort

cargo add glidesort
Rust specifics
Unwinding panics are Rust’s billion dollar mistake
Corporate needs you to find the differences between this picture and this picture.

They're the same picture.
Unwinding panics & generic unsafe code

1. Foreign code? Any call can cause unwinding.
2. Safe function? Have to be sound even during unwinding.
3. All traits are foreign code.
Unwinding panics & generic unsafe code

Writing non-trivial generic unsafe code is a nightmare.
All algorithm state in structs with Drop handlers.
~10-15% performance penalty in Glidesort for integers (can’t even panic!)
A great strength

- Moves are `memcpy`s, no move constructor!
- Makes optimizations possible:
  ```cpp
  // ptr::copy(scan, if less { dest } else { scratch }, 1);
  ptr::copy(scan, dest, 1);
  ptr::copy(scan, scratch, 1);
  ```
- Opposite of unwinding panics: no surprises.
Safe concatenation

- \([T]\text{::split\_at\_mut}\) is a one-way street
- Glidesort needs concatenation...
- Raw pointers?
- Branded slices!

GhostCell: Separating Permissions from Data in Rust
Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, Derek Dreyer.
pub struct MutSlice<'l, B, T, S> {
    begin: *mut T,
    end: *mut T,
    _lifetime: PhantomData<&'l mut T>,
    _metadata: PhantomData<(B, S)>,
}

---

Brand

State

unsafe upgrade()
assume_uninit()
unsafe assume_init()
unsafe take()

Example transitions

All slices within Glidesort are MutSlices!
I’m leaving academia

I’m open to interesting* (Rust) jobs!

https://orlp.net

https://linkedin.com/in/orson-peters

*I am not interested in cryptocurrency, Web3 or similar ventures.
Glide.