Effizientes Programmieren in C, C++ und Rust

A Short Introduction to Parallel Programming

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

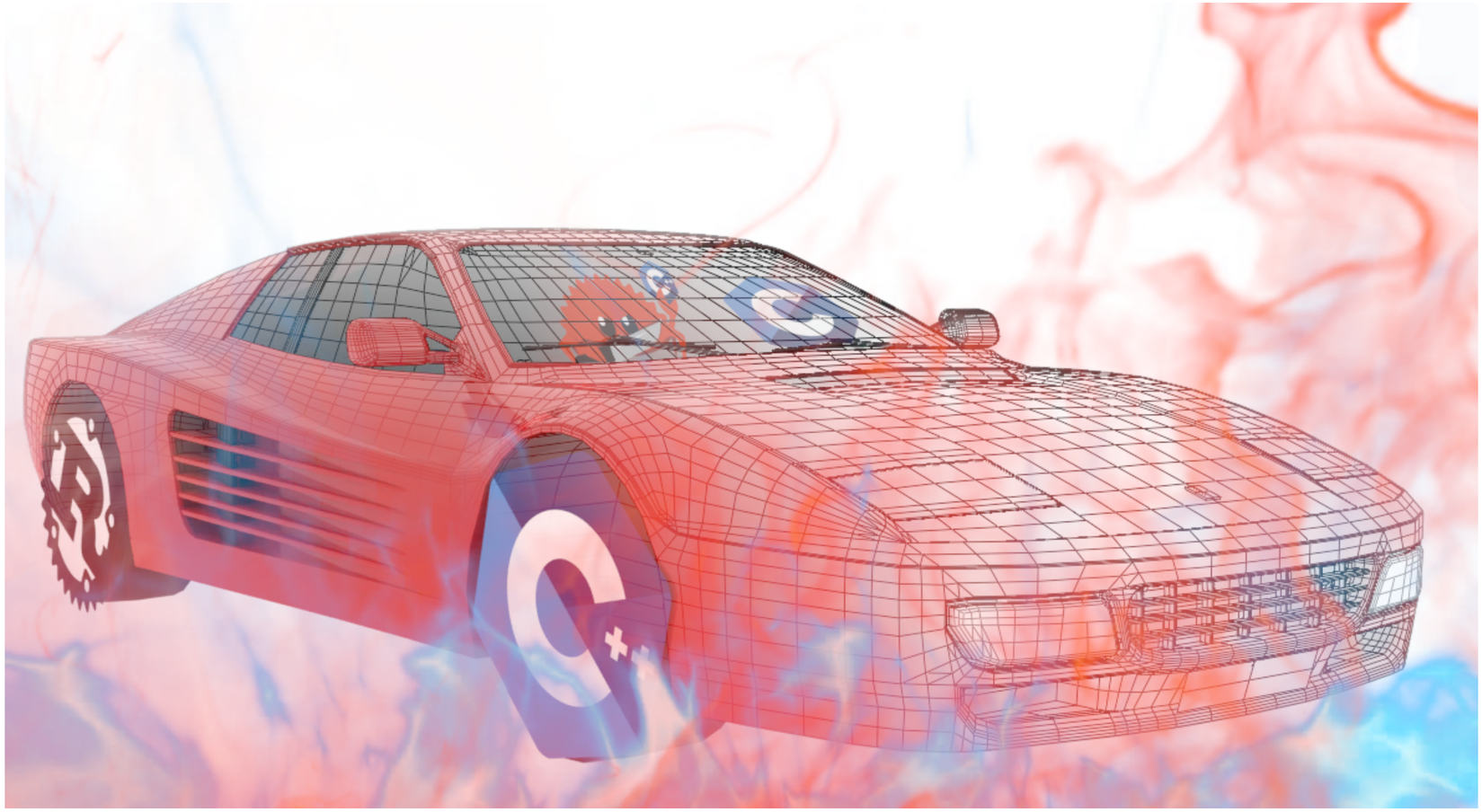


Table of Contents

Parallelism – Introduction

Why Parallelism?

- Keep main thread responsive by outsourcing work to other threads
  - *Very important for GUIs!*
- Get easy speedups by using some additional cores (typically order 2-8)
- Process large amounts of data/very difficult problems as fast as possible
  - *Wrong inference can only happen with `unsafe`*
  - *This is the most interesting and the most challenging*
- Single cores aren't getting much faster anymore

⇒ Parallelism is necessary to further improve performance 🚴

Challenges

- Correctness
- Scalability
- Synchronization Overhead

Data Races and Correctness

Nondeterminism

- Parallel program execution is fundamentally nondeterministic
- Different threads may be scheduled in any order
- Worse: there might not even be a global order; the world might look different for each thread
  - *The property that such a global order exists is known as **sequential consistency***

```
int racy_counter = 0;

auto count_to_hundred = [&]{
    for (int i = 0; i < 100; ++i) {
        ++racy_counter;
    }
};

std::thread first_thread(count_to_hundred);
std::thread second_thread(count_to_hundred);
first_thread.join();
second_thread.join();
```

Data Races

- Special case: concurrent access to the same data
- Data races are UB in both C++ and Rust
  (while other nondeterminism might just cause logic bugs)

Data Races and Correctness

Borrow Checking to the Rescue!

- Data races can not happen if mutable access is exclusive
- Bonus: borrow checking ensures that references are still alive
- However, general race conditions and deadlocks are still possible

Exception: Interior Mutability

- Interior mutability is used to construct explicit multi-threaded types
  (e.g. `AtomicX` and `Mutex`)
- But what about non-threadsafe types like `Rc` and `Cell`?

```
let mut racy_counter = 0;

let mut count_to_hundred = || {
    for _ in 0..100 {
        racy_counter += 1;
    }
};

thread::scope(|s| {
    //  ^^^^^ this API ensures that the threads
    //        don't outlive the current function
    s.spawn(&mut count_to_hundred);
    s.spawn(&mut count_to_hundred);
});
```

error[E0499]: cannot borrow `count_to_hundred` as mutable more than once at a time

Data Races and Correctness

Send and Sync

- Rust uses marker traits for thread safety
- `Send`: allows moving the type to another thread
- *Sending exclusive references is equivalent to* ***Send***
- `Sync`: allows sending shared references to other threads
- Auto traits: being `Send`/ `Sync` is automatically inferred from members
  - Wrong inference can only happen with `unsafe`

- `Cell`: `Send` but not `Sync`
- `Rc`: neither `Send` nor `Sync`

```
let cell = Cell::<i32>::new(0);
let shr_ref = &cell;
thread::scope(|s| {
    s.spawn(move || {
        shr_ref.set(shr_ref.get() + 1);
    });
});
```

error[E0277]: `Cell<i32>` cannot be shared between threads safely

```
let mut cell = Cell::<i32>::new(0);
let exl_ref = &mut cell;
thread::scope(|s| {
    s.spawn(move || {
        exl_ref.set(exl_ref.get() + 1);
    });
});
```

- Fine (no other thread can access it!)

Scalability

- Motivation: achievable speedup
- Example: Which is better?
  1. Speedup of 4 with 4 threads, speedup of 5 with 64 threads
  2. Speedup of 2 with 4 threads, speedup of 30 with 64 threads
- Answer: It depends, but 2. is more scalable

$\Rightarrow$ We need asymptotic analysis that includes number of threads p

Example: Naive Parallel Quicksort

```
fn quicksort_naive(data: &mut [u32]) {
    let left, splitter, right = partition(data);
    run_in_parallel(
        quicksort(left),
        quicksort(right),
    );
}
```

Note: this is not Rust, but pseudocode with Rust syntax

- Bottleneck: sequential partition
- Expected running time:

$$\mathcal{O}(n + \frac{n \log n}{p})$$

$\Rightarrow$ At most $\log n$ speedup
(and not a great constant factor)

$\Rightarrow$ Not scalable

Collective Operations

- Intended for distributed programs, but generally useful for parallel algorithms
- E.g. broadcast, gather, reduce, all-to-all

Prefix Sums

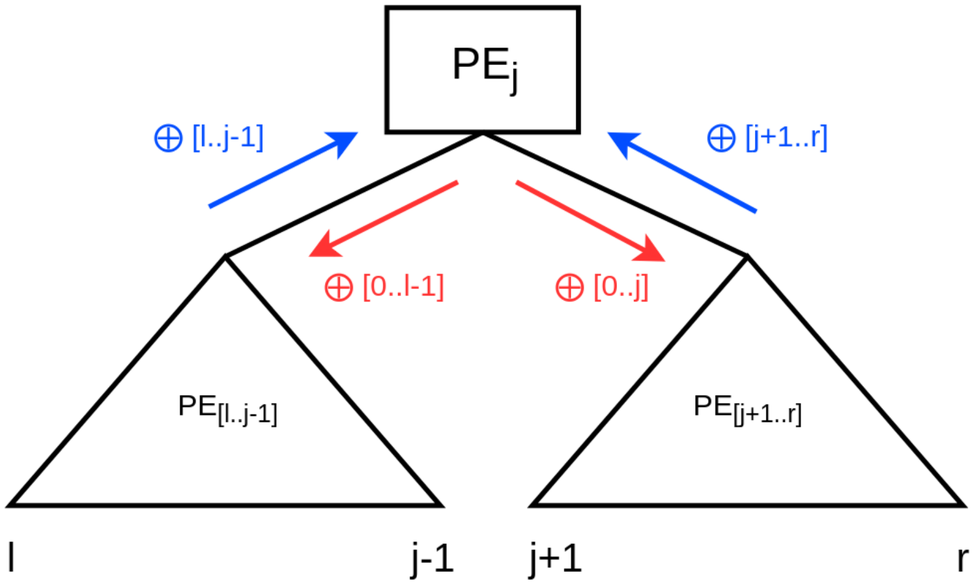
- Given: array of size n with entries a_i
- Goal: new array (or overwrite) such that

$$b_i = \bigoplus_{j=1}^{i} a_j$$

for some associative operation $\oplus$

$\Rightarrow$ Possible in $\mathcal{O}(\frac{n}{p} + \log p)$

(assuming elements of constant size)



[David Schneider, https://en.wikipedia.org/wiki/prefix_sum]

- Prefix sums are everywhere
- Central building block for parallel algorithms doing some kind of collective computation

Scalability and Parallel Algorithms

Scalable Sorting

- Parallelize the partitioning step

  $\Rightarrow$ Prefix sum for new position

- Expected running time:

$$\mathcal{O}\left(\frac{n \log n}{p} + \log^2 p\right)$$

- Scalable!
- Disadvantage: not in-place 👎

- *More of this in the lecture "Parallele Algorithmen" ↑*

```rust
fn quicksort_scalable(data: &mut [u32]) {
    let left, splitter, right =
        parallel_partition(data);
    run_in_parallel(
        quicksort(left), quicksort(right)
    );
}
```

```rust
fn parallel_partition(data: &mut [u32]) {
    let splitter = /* determine splitter */;
    let cmp = vec![0, data.len()];
    parallel for (i, val) in data {
        cmp[i] = if val <= splitter { 1 } else { 0 };
    }
    let index_sum, total = parallel_prefix_sum(cmp);
    let sorted = vec![0, data.len()];
    parallel for (i, val) in data {
        let new_index = if cmp[i] {
            index_sum[i]
        } else {
            total + i - index_sum[i]
        };
        sorted[new_index] = val;
    }
    sorted[0..total], splitter, sorted[total..]
}
```

Load Balancing

- Goal: divide work evenly between threads, even for complex workloads

WORK STEALING

- Assumption: tasks are divisible into smaller task
- Each thread maintains a local task queue
- If empty, steal from other threads

👍 "Silver bullet": good performance for almost every workload

👍 Flexible (dynamic task queuing)

👎 Quite complex

STATIC LOAD BALANCING

- Assign all work at the beginning
- Randomize in case of unpredictable workloads

👍 No overhead, simple

👎 Inflexible, imperfect balance

MASTER/WORKER

- Central thread (master) distributes work to worker threads

👍 Simple and very flexible

👎 Not scalable!

Synchronization Overheads

- Starting and stopping threads has notable overhead

  $\Rightarrow$ Always use thread pool

- Threads need to synchronize via atomic operations
  - *Note: mutex, message queues, join etc. are all based on atomics*
- Problem: most caches are thread-specific, hardware needs to transfer data (aka *cache coherency*)

  $\Rightarrow$ Concurrent accesses are much more expensive than regular memory access
  (only reading is okay, no transfer needed)

Modeling Synchronization

- One option: introduce additional cost factor α for every data exchange

False Sharing

- Hardware operates on cache lines
- Two threads might access disjoint data that happens to be on the same cache line

  $\Rightarrow$ Massive slowdown though access is semantically not concurrent
- Solvable by enforcing alignment manually