Lab C

CPU used:

- AMD Ryzen 5 5500U
  - 6 core/12 threads

For all the benchmarks we used: futhark bench, to do our benchmarks

Exercise 1

Testing process and process_idx

Given two input lists `s1` & `s2` such that:

```
s1 = [23,45,-23,44,23,54,23,12,34,54,7,2, 4,67]
s2 = [-2, 3,  4,57,34, 2, 5,56,56, 3,3,5,77,89]
```

process outputs:

```
73i32
```

and process_idx outputs:

```
73i32
12i64
```

Benchmarking results

By using the multicore backend we can see that the speedup (compared to when using the c backend) scales with the input size for both `process` and `process_idx`. As can be seen in *Fig 1* and *Fig 2*, both `process` and `process_idx` scale in a similar manner, that is, both scale linearly.

Graphs

Data generated by futhark:

two_%_i32s:

futhark dataset -b --i32-bounds=-10000:10000 -g [$*]i32 -g [$*]i32 > $@

With the sizes :

100, 1000, 10000, 100000, 1000000, 5000000, 10000000

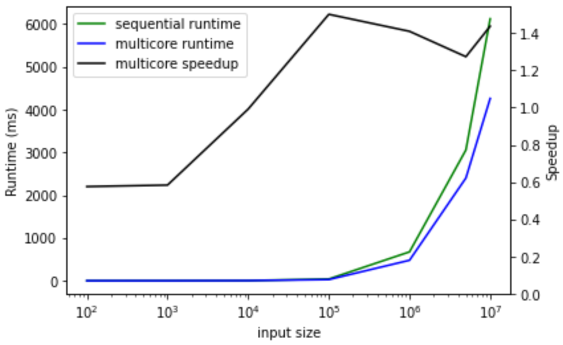

Fig 1: process runtime and speed up on multicore compared to c

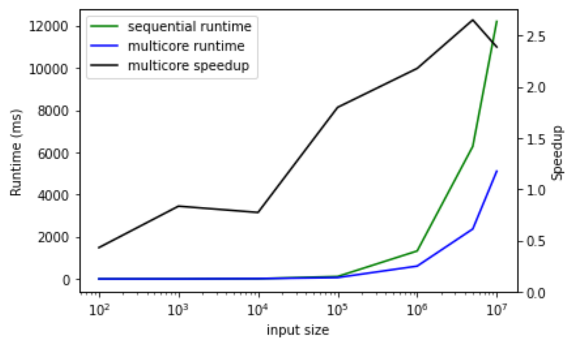

Fig 2: process_idx runtime and speedup on multicore compared to c

Exercise 2

Proof of left-identity

Here we prove that (0, false) is a left-neutral element of:

$$(v1, f1) \oplus' (v2, f2) = (\texttt{if } f2 \texttt{ then } v2 \texttt{ else } v1 \oplus v2, f1 \vee f2)$$

$(0, \text{false}) \oplus' (v, f)$	=	$(\texttt{if } f \texttt{ then } v \texttt{ else } 0 \oplus v, \text{false} \vee f)$	(0 is neutral for $\oplus$ and false is neutral for $\vee$)
	=	$(\texttt{if } f \texttt{ then } v \texttt{ else } v, f)$	
	=	(v, f)	

■

Benchmarking results for segscan and segreduce

The function we benchmark is addition (+).

When benchmarking `segscan`, `segreduce`, `scan` and `reduce` we used both the c backend and the multicore backend. In *Fig 4* and *Fig 5* we can see how our `segscan` and how the built-in `scan` compares and in *Fig 8* and *Fig 9* we can see how our `segreduce` compares to the built-in `reduce`. In both cases we get slowdowns which is not surprising since more work is done in `segscan` and `segreduce`.

Both `segscan` and `scan` have the same scaling. This is also the case for `segreduce` and `reduce`: That we have linear scaling but `reduce` is just an order of magnitude faster, which makes it look like constant scaling in Fig 8 and 9. One reason for this may be that we use addition as the operator, because of this futhark uses `reduce_comm`[1] instead of `reduce` when benchmarking and `reduce_comm` is faster. However, both scale linearly with the input, which can be seen in *Fig 6* and *Fig 7* (note log scaled x-axis).

---

[1] https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#4953

Graphs

Data generated by futhark:

For built-in:
one_%_i32s:

```
futhark dataset -b --i32-bounds=-10000:10000 -g [$*]i32 > $@
```

For segmented:
i32_%_bools:

```
futhark dataset -b --i32-bounds=-10000:10000 -g [$*]i32 -g [$*]bool > $@
```

Both with the sizes :

100, 1000, 10000, 100000, 1000000, 5000000, 10000000

Segscan

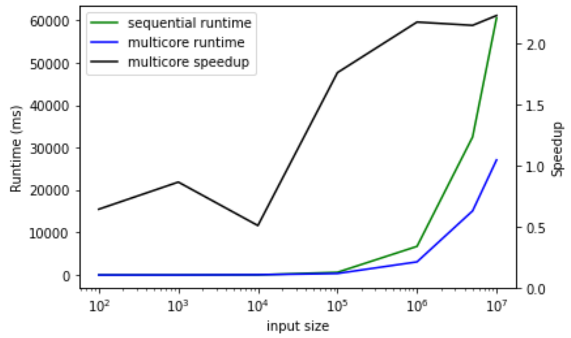

Fig 3: segscan runtime and speed up on multicore compared to c

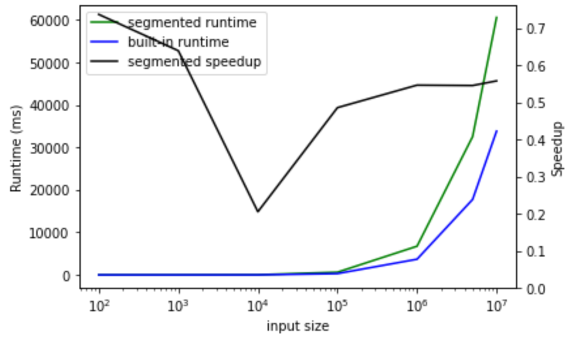

Fig 4: segscan compared to scan on c

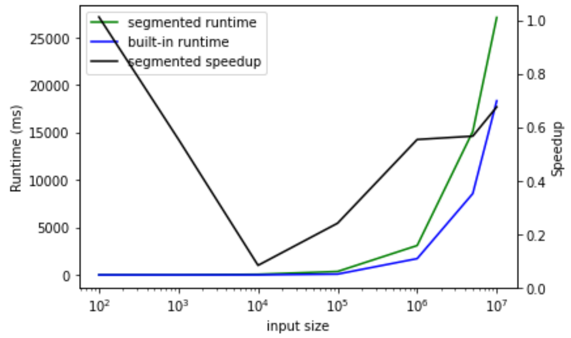

Fig 5: segscan compared to scan on multicore

Segreduce

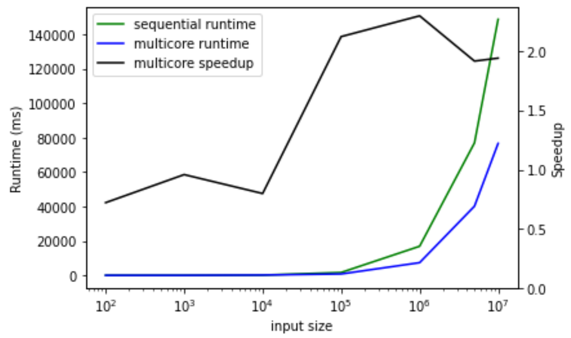

Fig 6: segreduce runtime and speedup on multicore compared to c

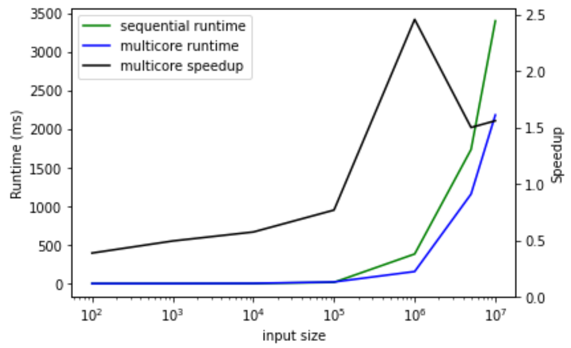

Fig 7: reduce runtime and speedup on multicore compared to c

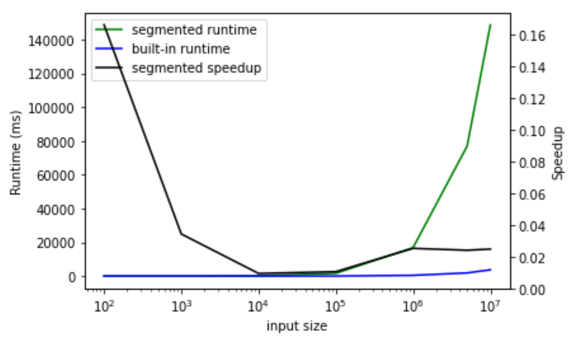

Fig 8: segreduce compared to reduce on c

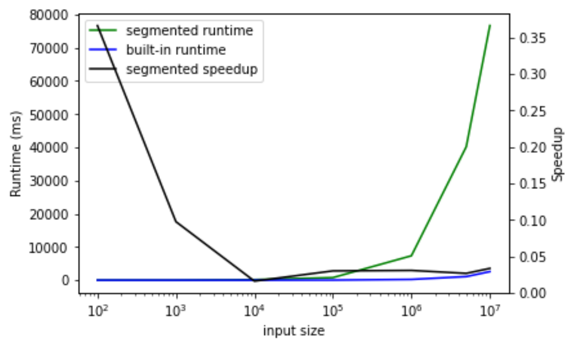

Fig 9: segreduce compared to reduce on multicore

Asymptotic complexity of our reduce_by_index

The two most expensive operations it uses are `radix_sort_by_key` and `segreduce`. The work and span complexity for `radix_sort_by_key` is *O(k n)* and *O(k log n)*[2] respectively, where k is constant. To get the complexity of `segreduce` we first need the complexity of `segscan`.

Both `segscan` and `scan` have the same complexity. The complexity of `scan` (which is the only operation used in `segscan`) has a work and span complexity that depends on the operator used in the function. In the documentation the work is written as *O(n W(op))* and the span is written as *O(log(n) W(op))* where *W(op)* is the work complexity for the operator[3].

The most expensive operation of `segreduce` is `segscan` and thus the complexity for `segreduce` is the same as the complexity for `segscan`.

Going back to the work and span complexity of `radix_sort_by_key`; Since k is constant and W(op) can be of any complexity we can conclude that the complexity for our `reduce_by_index` is:

- Work: *O(n W(op))*
- Span: *O(log(n) W(op))*

If we compare it to the built-in `reduce_by_index` who has[4]:

- Work: *O(n W(op))*
- Span:
  - Worst case: *O(n W(op))*
  - Best case: *O(W(op))*

Benchmarking results for reduce_by_index

The function we benchmark is addition (+).

---

2 https://futhark-lang.org/pkgs/github.com/diku-dk/sorts/0.4.1/doc/lib/github.com/diku-dk/sorts/radix_sort.html#abstract

3 https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#5011

4 https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#4967

Our `reduce_by_index` worked by first sorting and then doing a segmented reduction. This should in practice be better to do than the built-in `reduce_by_index` when we have many millions of bins[5]. But in practice, our solution is much slower than the built-in variant.

As with `segreduce`, our function scales like the built-in one, it is just a lot slower. We can see, when comparing *Fig 10* and *Fig 11*, that both our implementation and the built-in one scales linearly.

Data generated by futhark:

one_%_two:

```
futhark dataset -b --i32-bounds=-10000:10000 \
                   --i64-bounds=-10:$* \
                        -g [$*]i32 -g [$*]i64 -g [$*]i32 > $@
```

With the sizes :

100, 1000, 10000, 100000, 1000000, 5000000, 10000000

Graphs

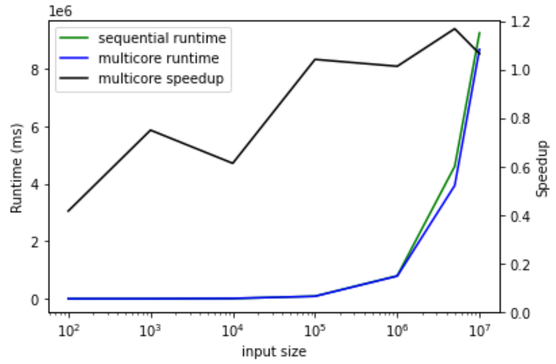

Fig 10: reduce_by_index runtime and speedup on multicore compared to c

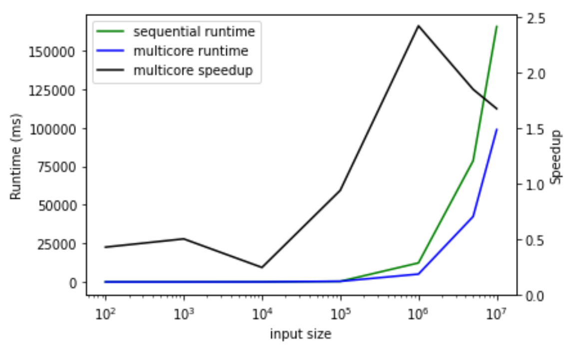

Fig 11: built-in reduce_by_index runtime and speedup on multicore compared to c

[5] https://futhark.readthedocs.io/en/latest/performance.html#histograms

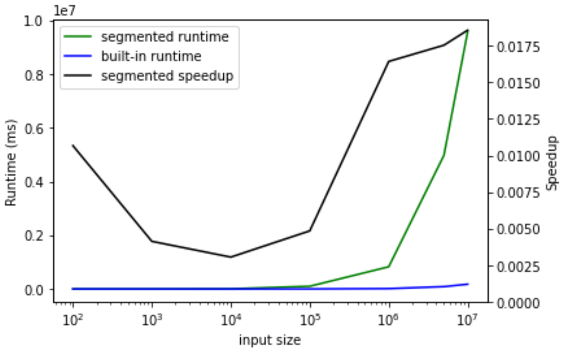

Fig 12: reduce_by_index compared to built-in reduce_by_index on c

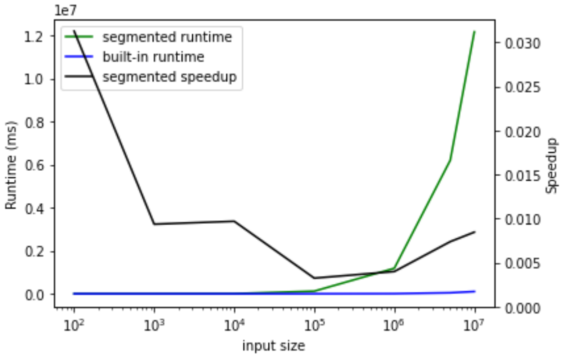

Fig 13: reduce_by_index compared to built-in reduce_by_index on multicore

Exercise 3

Benchmarking results

parameter	***default value***
h,w	*400*
n	*20*
abs_temp	*0.5*
samplerate	*0.1*

parameters	***sizes we test***
w,h,h and w	*100, 200, 400, 800, 1600*
n	*10,20,40,80*

Graphs

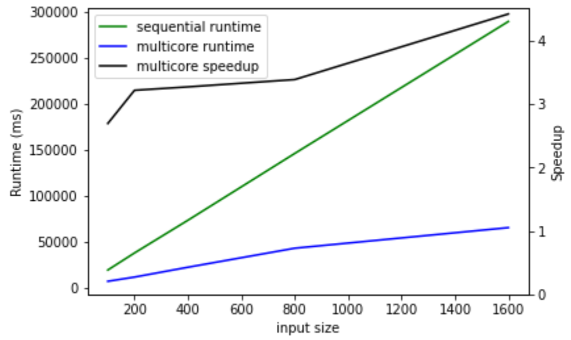

Fig 14: x-axis: size of h

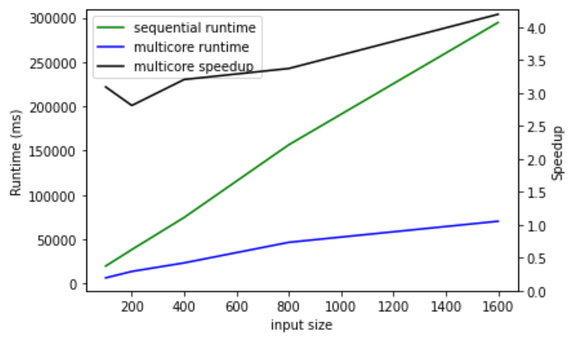

Fig 15: x-axis: size of w

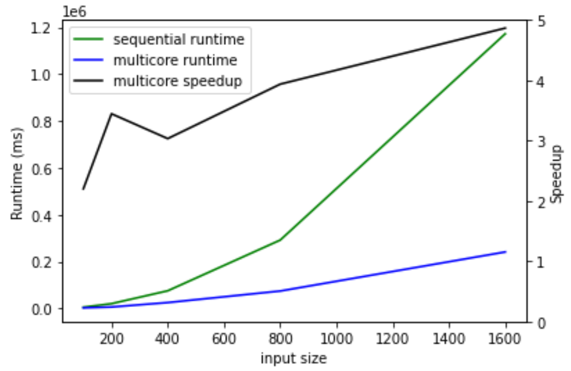

Fig 16: x-axis: size of w and h

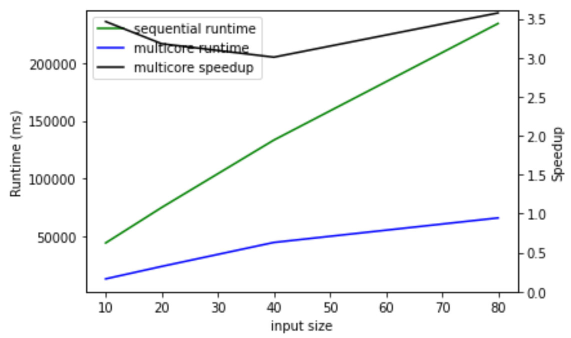

Fig 17: x-axis: size of n

Conclusion

We got linear scaling when we varied either w, h or n and quadratic when we varied both w and h at the same time, as one would expect. We can also see that we got a good speedup (between 3-5x) and this is without thinking about creating a parallel program. We just needed to change the backend to multicore and our solution was parallized.

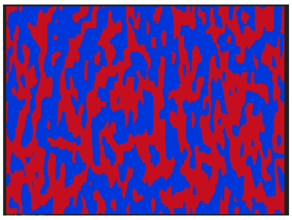

Fig 18: visualization of the ising model