

## PPA-F25 hw2

### Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There are 5 tasks: one coding task and four written tasks.

Please submit your work on Gradescope.

Create a file `written.pdf` with your solutions to the written tasks. Typeset solutions are preferred. Handwritten solutions will be accepted only if they are **easily legible**.

For the coding task, you will edit and submit `LongestLine.sml`.

**Due Date:** This submission is due at **5:00pm EDT on Monday, Sep 22**. Course policy on late submissions is available on the course website. (<https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/>)

### Setup

- Follow the instructions on the course website to access one of the **crunchy** compute servers.
- Install MaPLe. For the **crunchy** servers, you can:
  - Download <https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/resources/mpl-845f76f.tgz> and copy it to the server
  - Unpack by running `tar xzf mpl-845f76f.tgz`. This will create a directory `mpl-845f76f/`
  - The compiler, `mpl`, is located at `mpl-845f76f/bin/mpl`
  - Add `mpl-845f76f/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-845f76f/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

### Longest Line

Here you will use your reduction skills to compute the longest line of a text file (both the line number, and the length of that line). This is easy to do sequentially in  $O(n)$  work with just a few lines of code. In parallel, however, the solution isn't so obvious. The challenge is that newline characters might be arbitrarily far away from each other. For example, the longest line of a file might be millions of characters long. Any solution which sequentially walks between newline characters is doomed to have high span (and therefore not much parallelism) in the worst case.

It turns out that you can solve this problem (work-efficiently, with low span) using just a single parallel reduction across the whole input.

**Task 1 (60 points):** In `LongestLine.sml`, implement the function

`longest_line(chars)`. The output should be a tuple  $(i, k)$  where  $i$  is the line number of the longest line, and  $k$  is the length of the longest line.

Some requirements/notes/thoughts:

- Your solution must have worst-case  $O(n)$  work and  $O(\log n)$  span, where  $n$  is the length of the input sequence.
- If there are multiple lines that are all tied for the longest, then you should return the first longest line (the minimum line number).
- Your solution should contain exactly one call to the function `Parallel.reduce`, ranging over the entire input sequence. (Of course, the output of your `reduce` might not directly solve the problem; you are permitted to do other computation afterwards as long as your solution still meets the required work and span bounds.)
- Note that, by convention, line numbers are indexed from 1 instead of 0. For example, on the input `1\n123\n12`, the longest line is line number 2.
- **Hint:** review the lecture notes on the Maximum Contiguous Subsequence Sum problem.
- **Hint:** Sam's solution strengthens the problem into a 5-tuple.
- The image in Figure 1 is presented without comment. Maybe it is helpful.
- We recommend first thinking about solving a simpler problem: identifying only the *length* of the longest line. After you solve this problem, then think about how you can strengthen your solution to also compute the line number of the longest line.
- When designing the “combine” function for your reduction, **be careful about associativity**. In the past, students' submissions have been subtly wrong, with associativity issues that only appeared in rare cases. When testing, we will use multiple implementations of `Parallel.reduce` (that differ in the order of application of the combine function).

The syntax for a reduction is as follows.

```
Parallel.reduce g z (lo, hi) f
```

Here, the arguments are:

```
g: 'a * 'a -> 'a      (* (for any type 'a) an associative "combining" function *)
z: 'a                  (* an identity element for `g` *)
lo: int, hi: int       (* range of indices for `f` *)
f: int -> 'a           (* a function to generate one element of the reduction *)
```

Altogether, Task 1 is worth 60 points. This will be graded in two parts: 30 points for the correctness of the longest line number, and 30 points for the correctness of the longest line length. So, in other words, you can get half credit on this task by always correctly computing the length of the longest line but returning a bogus value for the line number.

**Testing.** A few test cases are included at the top of `test.sml`. Feel free to add more. You can test your code as shown below. If you change `test.sml`, you will need to recompile.

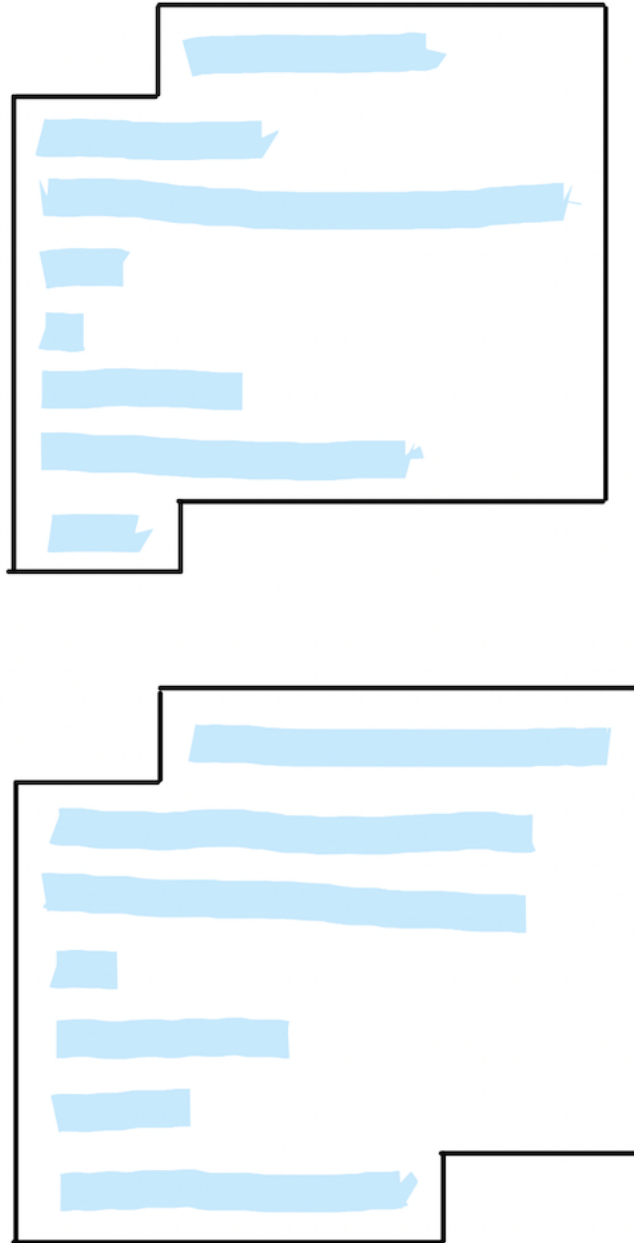


Figure 1: Perhaps a hint for the longest line problem?

```
$ make test
$ ./test
```

**Task 2 (5 points):** Using the commands shown below, download the list of words and run your solution to find the longest word. (Each word in this file is on its own line.) What is the longest word in the list?

```
$ ./download_words_dataset.sh # creates a file: data/words.txt
$ make main
$ ./main @mpl procs 10 -- data/words.txt
```

## Mergesort with Reduce

Suppose you have a function `merge: int Seq.t * int Seq.t -> int Seq.t` where `merge(S,T)` takes two *sorted* sequences `S` and `T` as argument and merges them, producing an output sequence that is sorted and contains all of the input elements of both `S` and `T`. For example, `merge([0,5,6],[1,2,3,4,5,6,7])` produces the sequence `[0,1,2,3,4,5,5,6,6,7]`.

We can use `merge` to implement a parallel mergesort as follows.

```
fun mergesort(X: int Seq.t) =
  if Seq.length(X) <= 1 then
    X
  else
    let
      (* split the sequence in half, producing two sequences, L and R *)
      val (L, R) = Seq.split_middle(X)
      val (sortedL, sortedR) =
        ForkJoin.par (fn () => mergesort(L),
                      fn () => mergesort(R))
    in
      merge(sortedL, sortedR)
    end
```

If `S` is length  $n$  and `T` is length  $m$ , then `merge` has  $O(n+m)$  work and  $O(\log^2(n+m))$  span. (This is possible in practice and later in the course we will look closely at how you can implement `merge` within these cost bounds.) Assume that `Seq.split_middle` and `Seq.length` both require  $O(1)$  work and span.

**Task 3 (10 points).** Write a recurrence  $W(n)$  for the work of this mergesort implementation, given in terms of  $n$ , the length of the input sequence. Just write the recurrence; you don't need to solve it.

**Task 4 (10 points):** Now write a recurrence  $S(n)$  for the span of mergesort, again in terms of  $n$ , the length of the input sequence. Just write the recurrence; you don't need to solve it.

**Task 5 (15 points):** Redefine `mergesort(X)` in terms of `Parallel.reduce`. (Include this in your submitted `written.pdf`.) The solution should have exactly

one call to `Parallel.reduce` and should only require at most a few lines of code.

Recall that `Parallel.reduce g z (lo, hi) f` executes `f(lo), ..., f(hi-1)` in parallel and combines all of the results using the associative function `g` and corresponding “identity” `z`.

In this task you can use functions such as:

- `Seq.length(X)` which returns the length of sequence `X`.
- `Seq.nth X i` which returns the  $i^{\text{th}}$  element of `X`.
- `Seq.singleton(x)` which returns the “singleton” sequence (of length 1) containing just the element `x`.
- `Seq.empty()` which returns an empty sequence (length 0).