Exam 2

Edgar Andrés Margffoy Tuay 201412566

Universidad de los Andes Concurrency, Parallelism and Distribution

30 de octubre de 2018

1. MergeSort: Scala and Elixir implementation comparison

With respect to Listing 2, the Scala implementation of mergesort (Listing 1) is equivalent, as it uses Fork-Join to paralelize the splitting and merging of each sublist (Left and Right), while it sorts a list sequentially when its length is lesser or equal than the cutoff preset value. In contrast, the naïve implementation of mergesort on Elixir, as shown on Listing 3 spawns a process for each split of a list, until each split has a single element. In all three implementations, the merge pass is done over the same process that forked the process for sorting the left sublist.

1.1. Parallel Complexity

If the merge pass is not parallelized for both Elixir implementations, then the work complexity would be $\mathcal{O}(n)$, whereas the span complexity would correspond to $\mathcal{O}(logn)$. In contrast, the parallel complexity using binary search-based merge would account for an overall complexity of:

- Span: $\mathcal{O}(\log^3(n))$
- Work: $\mathcal{O}(n\log(n))$

With respect to the Scala implementation, as it does the merge pass on a sequential fashion, its work complexity corresponds to $\mathcal{O}(n)$, while its span corresponds to $\mathcal{O}(log(n))$.

1.2. Could it be implemented on Elixir

As all lists and variables (Including parallel collections) are immutable on Elixir, the Scala implementation cannot be ported as-is, due to its reliance on memory manipulations (In-place modifications and memory copying) for optimal running time. Thus, there would be no time execution reduction if it was to be implemented on Elixir, *i.e.,* Listing 2.

```
    def sort(from: Int, until: Int, depth: Int): Unit = {
            if (depth == maxDepth) {
                    quickSort(xs, from, until - from)
            } else {
                    val mid = (from + until) / 2
                    val right = task {
                            sort(mid, until, depth + 1)
                    }
                    sort(from, mid, depth + 1)
                    right.join()

                    val flip = (maxDepth - depth) % 2 == 0
                    val src = if (flip) ys else xs
                    val dst = if (flip) xs else ys
                    merge(src, dst, from, mid, until)
            }
    }
```

Listing 1: Fork-Join implementation of Mergesort on Scala

```
    def sort(list) do
        if length(list) <= cutoff do
            Enum.sort(list)
        else
            mid = div length(list), 2
            {left, right} = Enum.split(list, mid)
            r_pid = Task.async(fn -> sort(right) end)
            left = sort(left)
            right = Task.await(r_pid)
            merge(left, right)
        end
    end
```

Listing 2: Fork-Join implementation of Mergesort on Elixir

```
def sort(list) do
    mid = div length(list), 2
    {left, risght} = Enum.split(list, mid)
    l_pid = Task.async(fn -> sort(left) end)
    r_pid = Task.async(fn -> sort(right) end)
    merge(Task.await(l_pid), Task.await(r_pid))
end
```

Listing 3: Naïve parallel implementation of Mergesort on Elixir