Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Santa Fe

Implementación de Métodos Computacionales
TC2037, Grupo 600

Gilberto Echeverría Furió

Actividad Integradora 5.3 Resaltador de sintaxis paralelo

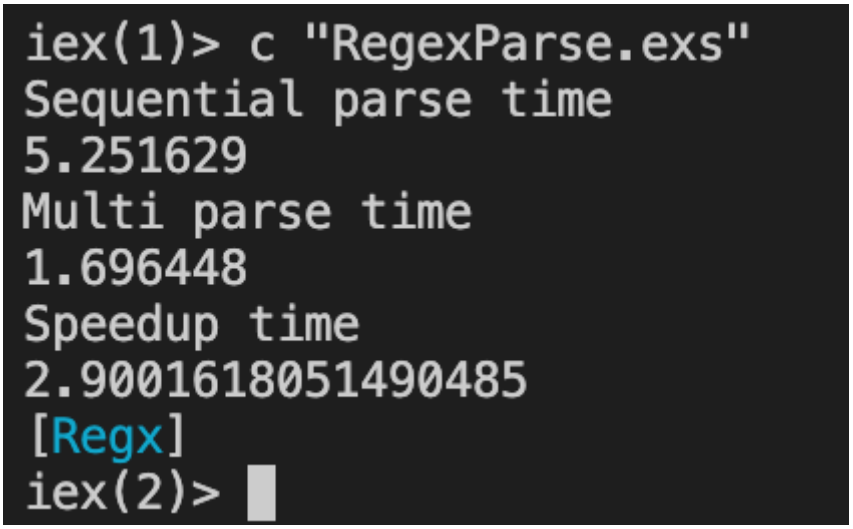
Equipo #2 | Integrantes:

Juan Muniain Otero	| A01781341
Andrew Dunkerley Vera	| A01025076

Junio 2021

Execution time analysis

After executing both the sequential and the parallel functions anonymously within the timer and get_speedup functions, we obtained the following times after parsing every test file by mapping them and executing the parser either in a separate task, or purely sequentially by calling the main parser function for each file. The following values were returned:



It is possible to observe a decrease in execution time of approximately 3.5 seconds after the sequential function is called for every test time. Also, as for the speedup time, obtaining 2.9 seconds reflects an increase in performance by comparing it with the speed taken to process every file sequentially.

Complexity analysis

Considering our sequential approach to the parser, it can be noted that execution time sloley depends on the length of each file to be analyzed, which prompts an O(n) time complexity. By recursively operating each file after mapping it, the pipeline contained within the get_lines function also holds an O(n) complexity, as no other operations are required that could increase its complexity. Also, by implementing tail recursion within our token_from_line function that evaluates lines with regex expressions, space complexity is reduced to O(1) complexity as stack frames are disposed of in each step.

```
def get_lines(in_filename, out_filename) do
  expr =
    in_filename
    |> File.stream!()
    |> Enum.map(&token_from_line/1)
    |> Enum.filter(&(&1 != nil))
```

(File line reading recursive implementation)

As for every regex match/run operation contained within the token_from_line function, these hold an O(n) time complexity as the match depends on the length of the line being evaluated. By pattern matching to obtain the regex match tail, time complexity is still O(n), as a simple head the tail pattern match of the regex expression has O(1) complexity. The same holds for Regex.split operations, as the same pattern match of the regex evaluation of a line occurs.

```
(Regex.match?(~r/\s*\d+\.?\d*E?[+|-]?\d*\s*/,line)) ->
  [_string, token] = Regex.run(~r/(\s*\d+\.?\d*E?[+|-]?\d*\s*)/,line)
  [_h | t] = String.split(line, ~r/(\s*\d+\.?\d*E?[+|-]?\d*\s*)/,  parts: 2)
  tmp = "#{html_string}<span class='number'>#{token}</span>"
  html_string = tmp
  [tail] = t
  aftr = false
  token_from_line(tail,html_string,aftr,false)
```

(Regex matching implementation for digit values after keys)

The same holds for our parallel implementation within our multiParse function, however, it is important to note that by assigning an asynchronous task to the mapped files, time complexity then depends on the amount of files to be operated on, which results in an O(n)*k time complexity (with *k* representing the amount of files to be operated on).

```
  "./Test_HW/out_file_000017",
  "./Test_HW/out_file_000018",
  "./Test_HW/out_file_000019",
  "./Test_HW/out_file_000020"]
  |> Enum.map(&Task.async(fn -> get_lines(&1 <> ".json", &1 <> ".html") end))
  |> Enum.map(&Task.await(&1))
```

(Multi-file mapping and parallelism)

Conclusion

After implementing parallel programming to our parser, the improvement in execution time and latency is evident, taking into account that large files were processed in mere seconds. This, in turn with the complexity of our program, proves that parallel programming holds many uses in our daily lives, and is necessary for larger, more demanding programs. As seen in our program results, parallelism for document parsing can prove useful not only for research, but also manipulating data in an efficient manner while utilizing the full potential of computer processors.

However, it is necessary to maintain an ethical resolve by providing software solutions aimed towards making our lives easier, or fun. Malicious programs can make use of parallel programming, and given the increase and widespread availability of processors with multiple cores increases the odds of a surge in more sophisticated malware and malicious software.