Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Implementación de Métodos Computacionales
TC2037.600, Grupo 600

Gilberto Echeverría Furió

Actividad Integradora 5.3: Resaltador de sintaxis paralelo

Equipo #8 | Integrantes:

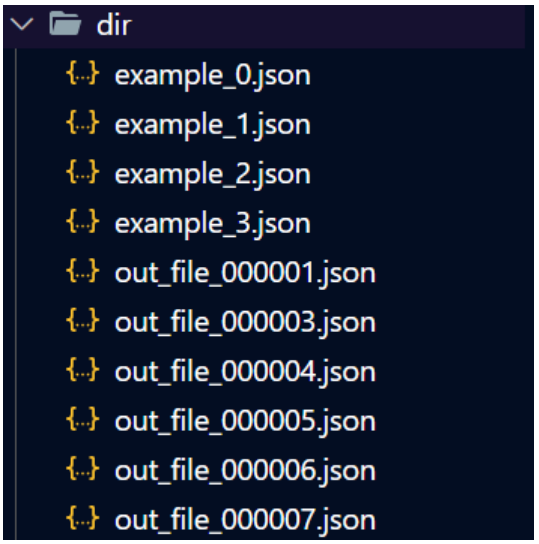
Miguel Ángel Bustamante Pérez	A00829919
Emilio Sibaja Villarreal	A01025139

Febrero 2021

Pruebas

Realizamos las siguientes pruebas:

Prueba 1



Resultados

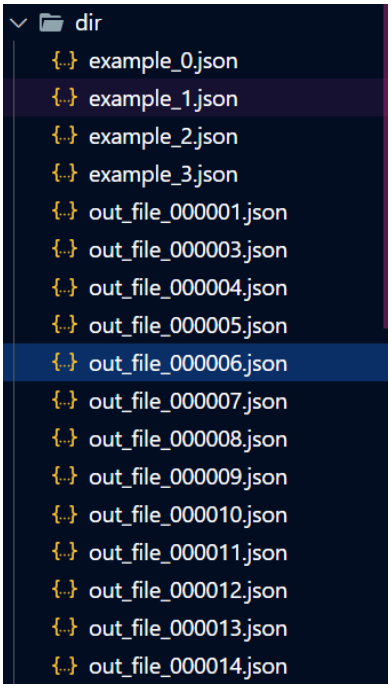
Paralelo: 6.45ms
Singular: 1.88s

Calculando el *speedup*:

$$speedup \ = \ time\ before\ upgrade\ /\ time\ after\ upgrade$$

speedup_prueba1 = 291.47

Prueba 2



Resultados

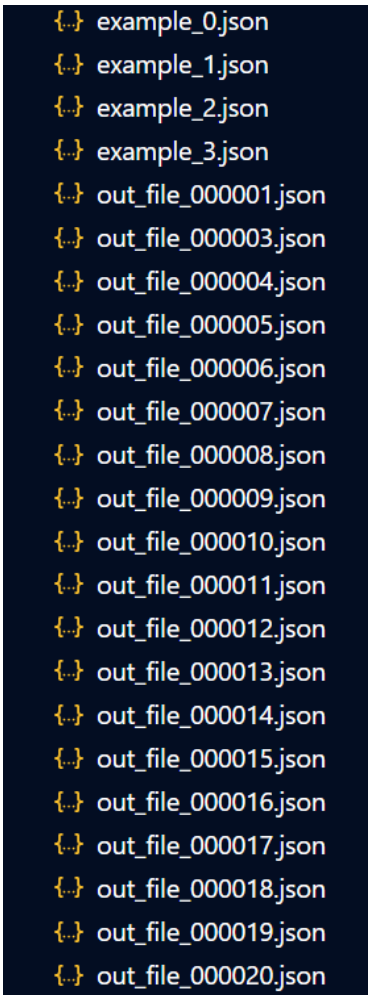
Paralelo: 7.47ms
Singular: 3.39s

Calculando el *speedup*:

$$speedup \ = \ time\ before\ upgrade \ / \ time\ after\ upgrade$$

speedup_prueba1 = 453.81

Prueba 3



Resultados

Paralelo: 6.24ms
Singular: 4.67s

Calculando el *speedup*:

$$speedup \ = \ time\ before\ upgrade \ / \ time\ after\ upgrade$$

speedup_prueba1 = 748.39

Complejidad

Para la parte singular, debido a que es secuencial depende de *n* que es el número de procesos a realizar, por lo que su complejidad es de:

$$O(n)$$

Para la parte paralela, está depende del número de archivos que el código va a procesar por lo que existe el multiplicador *x,* la complejidad por tanto queda de la forma:

$$O(n) * x$$

Conclusiones y reflexión final

Al realizar todas las pruebas y posteriormente calcular los *speedup* de cada una, es posible apreciar la inmensa diferencia que existe entre los tiempos de ejecución, puesto que los archivos empleados tienen miles y miles de líneas de texto para ser analizadas, siendo una tarea muy complicada para la forma singular, pero una muy sencilla de manera paralela.

Tomando en cuenta que un speedup que indica mejora debe ser mayor a uno, los resultados obtenidos superan con creces esa barrera, obteniendo valores mayores a 200, lo cual indica que para este tipo de problemas, es sumamente eficiente utilizar paralelismo en nuestro código. Al realizar varios procesos simultáneamente, el tiempo de ejecución de archivos grandes se reduce al mínimo.

Cabe destacar que el código que se utiliza para el análisis de archivos de texto debe de contener expresiones regulares que cumplan completamente con todos los posibles casos que se puedan encontrar dentro del archivo a procesar. En cuanto a este proyecto en particular fue necesario tener que pasar gran parte del tiempo de trabajo en el planteamiento y pruebas de los regex.

Estos fueron algunos de los casos que se utilizaron para probar parte de los archivos *.json* que en algunos momentos, funcionan mejor o peor.

Regex

Numbers(1):/(?<=:)[\d0-9+E.-]+(?!.*")/gm
Numbers(2):/(?=(?:[^"]*"[^"]*")*[^"]*\Z)(?=^) *[\d+E.-]/gm
Numbers(3):/-?[\d.]+(?:E?[+|-]?\d+)?(?!.*")/gm
Numbers(4):\s*-?[\d.]+(E?[+|-]?\d+)?(?!.*")\s

Keys(1): /(?=^) *\t*"[\w0-9:-]+" *(?=.*:)/gm
Keys(2): /(?=^)\t* *"[\w0-9:-]+" *(?=:)/gm
Keys(3):/ *\t*"[\w0-9:-]+" *(?=.*:)/gm

ReservedWord(1):/(?!.\d)(?<=:)[a-zA-Z]+(?!.*")/gm
ReservedWord(2):/(?!.*\d)(?=^)(?=(?:[^"]*"[^"]*")*[^"]*\Z) *[a-zA-Z]+/gm
ReservedWord(3): /(?!.\d)[null|true|false]+(?!.*")/gm

Es indispensable considerar casos de prueba simples similares a los archivos grandes, para evitar correcciones masivas en las expresiones regulares utilizadas en el código.