Comparación de Algoritmos: Implementación Secuencial en Python vs. Paralelización en Spark

Efraín García Valencia, Samuel Acevedo Bustamante, Nicolás Carmona Cardona

Resumen - Este informe detalla una comparación entre la implementación secuencial de un algoritmo en Python y su versión paralelizada utilizando Apache Spark. El objetivo es analizar las ventajas y desventajas en términos de eficiencia, tiempos de ejecución y uso de recursos del sistema, en el contexto del creciente interés por la paralelización en procesamiento de datos masivos.

Índice de Términos - Big Data, Paralelización, Apache Spark, Python.

I. INTRODUCCIÓN

El aumento exponencial en la generación de datos ha impulsado la necesidad de técnicas avanzadas para procesarlos de manera eficiente. Tradicionalmente, los algoritmos secuenciales han sido la base del procesamiento computacional, pero en la actualidad, su capacidad para manejar grandes volúmenes de datos se ha visto superada por las exigencias del entorno digital.

La computación paralela, que permite dividir tareas en múltiples hilos o procesos para ejecutarlas simultáneamente, se ha consolidado como una solución viable para superar estas limitaciones. Apache Spark, una herramienta diseñada específicamente para Big Data, ofrece capacidades avanzadas de paralelización a través de su modelo de Resilient Distributed Datasets (RDD). Por otro lado, Python sigue siendo una elección popular para el desarrollo de algoritmos debido a su simplicidad y extensa biblioteca.

Este proyecto nace de la motivación de explorar la eficiencia de estos enfoques, comparando una implementación secuencial en Python con su contraparte paralelizada en Spark. El objetivo es proporcionar una visión clara sobre cuándo y cómo utilizar cada enfoque, dependiendo del contexto y los recursos disponibles.

II. MARCO TEÓRICO

El marco teórico de este proyecto se centra en los principios y herramientas de la computación paralela, la gestión de recursos por parte de los sistemas operativos, y las características fundamentales de Apache Spark y Python como tecnologías de implementación.

A. Computación Secuencial vs. Paralela

La computación secuencial implica la ejecución de instrucciones una tras otra en un único núcleo de procesamiento. Aunque este modelo es adecuado para problemas simples o conjuntos de datos pequeños, enfrenta limitaciones significativas cuando se aplican a problemas de gran escala, especialmente en la era del Big Data.

Por otro lado, la computación paralela permite dividir un problema en subproblemas que pueden resolverse simultáneamente en múltiples núcleos o nodos. Este enfoque se fundamenta en conceptos clave como concurrencia, paralelismo y sincronización. La concurrencia se refiere a la capacidad de un sistema para manejar múltiples tareas al mismo tiempo, mientras que el paralelismo se centra en la ejecución simultánea de tareas para mejorar el rendimiento.

El modelo de computación paralela depende de la arquitectura del sistema subyacente y de la manera en que los recursos, como CPU y memoria, son gestionados por el sistema operativo. La efectividad del paralelismo puede verse limitada por factores como la sobrecarga de comunicación entre procesos, la sincronización y el balance de carga.

B. Rol de los Sistemas Operativos en la Computación Paralela

Los sistemas operativos son una pieza fundamental en la gestión de recursos para la computación paralela. Actúan como mediadores entre el hardware y las aplicaciones, garantizando que los recursos se distribuyan de manera eficiente y equitativa.

En entornos paralelos, los sistemas operativos gestionan:

1) Procesos e Hilos: Supervisan la creación, ejecución y finalización de procesos e hilos. Esto incluye la asignación de tiempo de CPU, memoria y otros recursos críticos.

2) Sincronización: Garantizan la coordinación entre procesos mediante mecanismos como semáforos, mutexes y monitores. Esto es esencial para evitar condiciones de carrera y garantizar la consistencia de datos compartidos.

3) Planificación de Tareas: Determinan qué tarea se ejecutará en qué momento, utilizando algoritmos como round-robin, prioridad o planificación basada en tiempo real.

4) Gestión de Memoria: Asignan y supervisan el uso de memoria, utilizando estrategias como la paginación o la segmentación para optimizar el acceso y minimizar conflictos.

Estas capacidades son esenciales para ejecutar tareas paralelas de manera efectiva, y su implementación puede variar significativamente entre sistemas operativos.

C. Apache Spark y RDDs

Apache Spark es una plataforma de computación distribuida diseñada específicamente para procesar grandes volúmenes de datos de manera rápida y eficiente. Su arquitectura se basa en un modelo maestro-esclavo, donde un nodo maestro coordina las tareas y varios nodos trabajadores ejecutan los cálculos. El elemento central de Spark es el concepto de Resilient Distributed Datasets (RDD), que son estructuras de datos distribuidas inmutables. Los RDDs permiten realizar operaciones paralelas de manera eficiente, gracias a características como inmutabilidad, tolerancia a fallos y soporte para transformaciones y acciones. Estas transformaciones incluyen operaciones como map y filter, que permiten manipular los datos distribuidos, mientras que las acciones, como collect o count, generan resultados concretos. Spark también cuenta con módulos avanzados como Spark SQL para consultas estructuradas y MLlib para aprendizaje automático, ampliando sus aplicaciones más allá del procesamiento básico de datos.

D. Python como Herramienta de Implementación

Python, por su parte, es un lenguaje de programación de alto nivel ampliamente reconocido por su simplicidad y extensibilidad. Su sintaxis intuitiva permite a los desarrolladores concentrarse en la lógica del algoritmo, reduciendo la complejidad del código. Esto lo convierte en una herramienta ideal para implementar algoritmos secuenciales. Además, la extensa biblioteca de Python incluye herramientas como NumPy y Pandas, que facilitan el análisis y procesamiento de datos, y Scikit-learn, que es ampliamente utilizado en aplicaciones de aprendizaje automático. Python también es altamente portátil, lo que le permite ejecutarse en múltiples sistemas operativos sin necesidad de modificaciones significativas. Sin embargo, debido a su naturaleza interpretada, Python puede presentar limitaciones de rendimiento en comparación con lenguajes compilados como C o Java. Estas limitaciones son más notorias en tareas computacionalmente intensivas, donde el tiempo de ejecución es crítico.

E. Desafíos en la Comparación

La comparación entre computación secuencial y paralela plantea desafíos únicos. Si bien la paralelización puede ofrecer ventajas significativas en términos de rendimiento y escalabilidad, también conlleva sobrecostos asociados con la configuración inicial y la sincronización entre procesos. Además, la efectividad de cada enfoque depende del tamaño de los datos y de las características del hardware y software utilizado. Por ejemplo, para volúmenes de datos pequeños, un enfoque secuencial puede ser más eficiente debido a la simplicidad de su configuración y ejecución. En cambio, cuando se trabaja con grandes volúmenes de datos, la capacidad de distribuir y procesar tareas en paralelo puede hacer que herramientas como Spark sean indispensables.

III. METODOLOGÍA

La metodología seguida en este proyecto se diseñó para permitir una comparación exhaustiva entre la implementación secuencial y paralelizada del algoritmo TF-IDF (Term Frequency-Inverse Document Frequency). Este enfoque asegura una evaluación justa y reproducible de ambas versiones del algoritmo, utilizando criterios claros y medibles como tiempo de ejecución, uso de recursos y escalabilidad.

El proceso comenzó con la selección del algoritmo TF-IDF debido a su importancia en aplicaciones de procesamiento de lenguaje natural (NLP), como la recuperación de información, clasificación de texto y minería de datos. Este algoritmo es ideal para analizar el rendimiento porque involucra tanto cálculos matemáticos intensivos como la manipulación de grandes volúmenes de datos, lo que lo hace representativo de los desafíos encontrados en Big Data y sistemas distribuidos.

La implementación se dividió en dos enfoques complementarios. En primer lugar, se desarrolló una versión secuencial en Python, aprovechando su simplicidad y extensas bibliotecas. Este desarrollo permitió establecer una línea base de rendimiento para comparar la paralelización. En segundo lugar, el algoritmo fue adaptado para Apache Spark, una plataforma diseñada para la computación distribuida y eficiente de grandes volúmenes de datos. Ambas versiones del algoritmo fueron validadas con pruebas iniciales para asegurar que produjeran los mismos resultados y fueran funcionalmente equivalentes.

El diseño experimental se estructuró para cubrir una variedad de escenarios. Se utilizaron conjuntos de datos sintéticos y reales con tamaños que varían desde pequeños documentos individuales hasta corpus de gigabytes. Las pruebas se realizaron bajo condiciones controladas, asegurando que las diferencias observadas fueran atribuibles únicamente a las características de cada implementación. Finalmente, los datos recopilados fueron analizados utilizando herramientas estadísticas y de visualización, proporcionando una

comprensión clara de las ventajas y limitaciones de cada enfoque.

IV. Implementación

La implementación del proyecto se desarrolló considerando tanto los aspectos de software como de hardware necesarios para garantizar la ejecución eficiente y confiable del algoritmo TF-IDF (Term Frequency-Inverse Document Frequency). Este proceso abarcó desde la preparación de los entornos de desarrollo y ejecución hasta el diseño y configuración de las pruebas experimentales.

El desarrollo del software involucró dos implementaciones del algoritmo: una secuencial en Python y otra paralelizada en Apache Spark. La versión en Python fue desarrollada para establecer una línea base de rendimiento, mientras que la versión en Spark se diseñó para explorar cómo la paralelización afecta la eficiencia y escalabilidad al manejar volúmenes de datos significativos. Cada implementación siguió un flujo bien definido y utilizó herramientas específicas para facilitar su desarrollo y prueba.

Para la implementación secuencial, se utilizó Python debido a su flexibilidad, simplicidad y extensa biblioteca de herramientas. Se emplearon librerías estándar como math para realizar cálculos matemáticos esenciales, y collections.Counter para contar la frecuencia de términos en los documentos. Estas herramientas simplificaron el desarrollo y aseguraron un rendimiento adecuado en escenarios secuenciales. Además, la biblioteca timeit se utilizó para medir con precisión los tiempos de ejecución, proporcionando datos confiables para la comparación.

La implementación paralelizada en Apache Spark aprovechó su capacidad para manejar grandes volúmenes de datos distribuidos entre múltiples nodos. Se utilizaron RDDs para realizar operaciones distribuidas como el cálculo de frecuencias de términos y la agregación de datos, mientras que los DataFrames facilitaron la manipulación estructurada de la información. Transformaciones como map y reduceByKey fueron fundamentales para dividir y consolidar datos en paralelo, optimizando el rendimiento en el clúster. Spark fue configurado en un entorno maestro-esclavo, donde el nodo maestro coordinaba las tareas distribuidas entre los nodos trabajadores, garantizando un uso eficiente de los recursos.

El clúster total incluyó 147 CPUs, 1230 GB de RAM y 121.2 TB de almacenamiento, distribuidos entre múltiples nodos con diferentes capacidades:

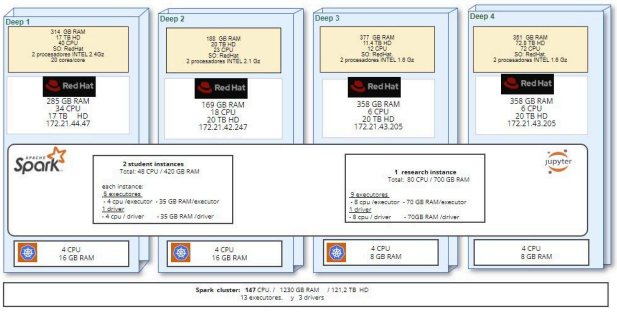
1) Deep 1: 40 CPUs, 314 GB de RAM, y 17 TB de almacenamiento.

2) Deep 2: 23 CPUs, 188 GB de RAM, y 20 TB de almacenamiento.

3) Deep 3: 12 CPUs, 377 GB de RAM, y 11.4 TB de almacenamiento.

4) Deep 4: 72 CPUs, 351 GB de RAM, y 72.8 TB de almacenamiento.

Cada nodo incluyó procesadores Intel con múltiples núcleos, optimizando las capacidades de procesamiento paralelo. Esta arquitectura permitió distribuir las cargas de trabajo del algoritmo TF-IDF, logrando un alto rendimiento en la extracción de características de la prueba con 50 documentos utilizados y luego con los 785 documentos utilizados.



Los datos de prueba consistieron en una prueba de 50 documentos de texto obtenidos del repositorio público Textfiles y luego con 785 documentos de texto obtenidos del repositorio público Textfiles. Este corpus incluyó una variedad de archivos, proporcionando un conjunto diverso para evaluar el rendimiento del algoritmo TF-IDF. Antes de la ejecución, los documentos fueron preprocesados para eliminar caracteres especiales, convertir el texto a minúsculas y tokenizar las palabras, asegurando una entrada limpia para ambos enfoques del algoritmo.

Las etapas importantes del algoritmo incluyeron:

1) Preprocesamiento de Texto: Eliminación de ruido y normalización de datos para garantizar una entrada uniforme.

2) Cálculo de Frecuencia de Términos (TF): Determinación de la relevancia local de las palabras dentro de cada documento.

3) Cálculo de Inverse Document Frequency (IDF): Evaluación de la relevancia global de las palabras considerando su presencia en todo el corpus.

4) Cálculo de TF-IDF: Combinación de TF e IDF para asignar pesos ponderados a cada palabra, destacando su importancia relativa en cada documento.

Los resultados se verificaron para confirmar que las implementaciones en Python y Spark producían resultados equivalentes, proporcionando una base sólida para el análisis comparativo, además, se verifica que el proyecto no solo cumpla con los objetivos planteados, sino que también sea un

aporte valioso para explorar la eficiencia de los algoritmos secuenciales y paralelizados en tareas de procesamiento de lenguaje natural.

V. Protocolo de Experimentación

El protocolo de experimentación fue diseñado para garantizar que los resultados obtenidos fueran reproducibles, confiables y representativos de las capacidades del algoritmo TF-IDF en sus versiones secuencial y paralelizada. Este diseño involucró estrategias específicas, una selección cuidadosa del conjunto de datos y el uso de herramientas adecuadas para la recolección, procesamiento y validación de datos.

A. Estrategias de experimentación

La experimentación se estructuró en tres etapas principales: preparación, ejecución y análisis. Durante la etapa de preparación, se configuraron los entornos de prueba, asegurando que las máquinas utilizadas para ejecutar el algoritmo tuvieran especificaciones homogéneas. Se eligió un enfoque iterativo, donde cada experimento fue ejecutado varias veces para reducir el impacto de posibles fluctuaciones en el rendimiento debido a factores externos. Estas iteraciones aseguraron que las métricas recopiladas fueran consistentes y estadísticamente significativas.

La ejecución de los experimentos se dividió en dos escenarios principales: uno para la implementación secuencial en Python y otro para la versión paralelizada en Apache Spark. Para garantizar una comparación justa, ambos enfoques utilizaron el mismo conjunto de datos y produjeron los mismos resultados funcionales. Se definieron métricas clave para evaluar los tiempos de ejecución.

Durante la etapa de análisis, los datos recopilados ayudaron a determinar cuál de las implementaciones resultaba más eficiente.

B. Conjunto de datos

El conjunto de datos utilizado en este proyecto consistió en una muestra de 50 documentos de texto extraídos de un repositorio público de archivos de texto con un total de 785 documentos. Este corpus es una fuente diversa que incluye manuales, literatura técnica y artículos históricos, ofreciendo un rango amplio de tamaños y complejidades. La diversidad de los documentos permitió evaluar cómo el algoritmo TF-IDF manejaba variaciones en el contenido y la estructura del texto.

Antes de la ejecución de los experimentos, los documentos fueron sometidos a un proceso de preprocesamiento que incluyó:

- Conversión a minúsculas para eliminar distinciones entre mayúsculas y minúsculas.
- Eliminación de caracteres especiales, números y puntuación para enfocar el análisis en las palabras significativas.
- Tokenización para dividir los documentos en palabras individuales, facilitando el cálculo de TF e IDF.

Este preprocesamiento garantizó que los datos ingresados al algoritmo estuvieran limpios y consistentes, reduciendo posibles sesgos o errores durante el cálculo.

C. Herramientas utilizadas

La implementación y experimentación se apoyaron en un conjunto de herramientas diseñadas para facilitar la ejecución, monitoreo y análisis del algoritmo en ambos enfoques.

1) Python: Utilizado para la implementación secuencial del algoritmo. Se emplearon librerías estándar como math para cálculos matemáticos y collections.Counter para la gestión de frecuencias de términos. Además, timeit fue clave para medir con precisión los tiempos de ejecución de esta versión.

2) Spark: Herramienta principal para la implementación paralelizada. Las transformaciones como map y reduceByKey permitieron dividir y consolidar los cálculos en el clúster. Las métricas internas de Spark proporcionaron datos detallados sobre el rendimiento, como el tiempo de ejecución y el uso de recursos en cada nodo del clúster.

3) Jupyter Notebook: Utilizado como entorno de desarrollo interactivo para documentar los experimentos, ejecutar scripts y visualizar resultados. Este entorno permitió integrar código, resultados y análisis en un solo lugar, facilitando la reproducibilidad del proyecto.

4) Hardware: El hardware utilizado consistió en un cluster diseñado para ejecutar tanto la versión secuencial como la paralelizada del algoritmo TF-IDF, garantizando condiciones óptimas de experimentación. Se usó la Research Instance, una máquina que incluye 80 CPUs y 700 GB de RAM, configurada para operar con 9 ejecutores. Cada ejecutor fue asignado con 8 CPUs y 70 GB de RAM, optimizando así la paralelización del algoritmo. Además, la instancia contó con un driver configurado con 8 CPUs y 70 GB de RAM, encargado de coordinar y distribuir las tareas entre los nodos del clúster. Los servidores en el clúster fueron gestionados con el sistema operativo Red Hat,

conocido por su robustez y estabilidad en entornos de procesamiento distribuido. Estas especificaciones garantizaron que ambos enfoques tuvieran suficiente capacidad para manejar los volúmenes de datos sin cuellos de botella significativos. La red del clúster, con baja latencia, facilitó la comunicación entre los nodos trabajadores, maximizando la eficiencia en Spark.

VI. Resultados

Ambos algoritmos (paralelizado y secuencial) fueron ejecutados con una muestra de 50 documentos para verificar su funcionalidad, posteriormente estos fueron ejecutados con los 785 documentos (el total de documentos del Dataset) para poner a prueba su eficiencia y la diferencia en sus tiempos de ejecución.

```
Timing Metrics:
1. Text Cleaning and Loading: 0.71 seconds
2. TF Calculation: 0.13 seconds
3. IDF Calculation: 1.10 seconds
4. TF-IDF and Keyword Extraction: 0.05 seconds
Overall Time: 1.99 seconds
```

Algoritmo paralelizado con 50 documentos

```
Timing Metrics:
1. Text Cleaning and Loading: 32.60 seconds
2. TF Calculation: 0.78 seconds
3. IDF Calculation: 0.78 seconds
4. TF-IDF and Keyword Extraction: 0.56 seconds
Overall Time: 34.72 seconds
```

Algoritmo secuencial con 50 documentos

```
Timing Metrics:
1. Text Cleaning and Loading: 0.79 seconds
2. TF Calculation: 0.17 seconds
3. IDF Calculation: 21.11 seconds
4. TF-IDF and Keyword Extraction: 0.04 seconds
Overall Time: 22.12 seconds
```

Algoritmo paralelizado con 785 documentos

```
Timing Metrics:
1. Text Cleaning and Loading: 440.66 seconds
2. TF Calculation: 11.40 seconds
3. IDF Calculation: 11.63 seconds
4. TF-IDF and Keyword Extraction: 10.81 seconds
Overall Time: 474.50 seconds
```

Algoritmo secuencial con 785 documentos

#Docs	50		785	
Algoritmo	NP	P	NP	P
Tiempo Ejecución	34.72s	1.99s	474.5s	22.12s
Incremento	-	-	13.66	11.11
Eficiencia P	-	17.45	-	21.45
Mejora en T	-	94.27%	-	95.34%

VII. Conclusiones

El uso de algoritmos y sistemas paralelizados aumenta la eficiencia de manera considerable para estos algoritmos repetitivos y que iteran sobre conjuntos de datos muy grandes.

El uso de clusters y herramientas especializadas en computación paralela hace que trabajar con la misma sea más sencillo y evita algunos de los problemas comunes como deadlocks.

VIII. Referencias

[1] B. Mustapic, “An introduction to TF-IDF: What is it and how to use it”, Semrush Blog, Feb. 28, 2024.[En línea]. Disponible en: https://www.semrush.com/blog/tf-idf/.
[2] “PySpark Overview”, Apache Spark. Sep. 09, 2024. [En línea]. Disponible en: https://spark.apache.org/docs/latest/api/python/index.html.