Cluster con Apache Spark

Vanessa K. Sotomayor A. [a] Jairo S. Valle R. [b] John J. Villavicencio S. [c]

[a] *San Cayetano alto 1101608, Loja, Loja, Ecuador*

[b] *San Cayetano alto 1101608, Loja, Loja, Ecuador*

[c] *San Cayetano alto 1101608, Loja, Loja, Ecuador*

Resumen

El presente documento explica de manera detallada como es el proceso de instalación e implementación del cluster Apache Sapark en Ubuntu, también se desarrolla una aplicación tanto en paralelo como en secuencial. Una vez que se tiene desarrollada la aplicación se realiza un plan de pruebas para obtener tiempo de ejecución, aceleración, eficiencia, y coste al ejecutar la aplicación en paralelo y secuencial, con esta información se realizará un análisis de rendimiento.

Key words: Cluster; BigData; Paralelo; Serial; Middelware; Spark; Grid Computing.

1. Introducción

En los últimos años la incorporación de avances de arquitectura en los microprocesadores ha sido significativa. Sin embargo, no podemos depender continuamente de procesadores más rápidos para obtener más eficiencia. Hay límites físicos, como la velocidad de la luz, que eventualmente van a desacelerar la reducción, que se ha visto año tras año, en el tiempo que dura un ciclo (tiempo para ejecutar la operación más básica) de CPU. Dadas las dificultades en mejorar la eficiencia de un procesador, la convergencia en eficiencia entre microprocesadores y los supercomputadores tradicionales, y el relativo bajo costo de los microprocesadores, ha permitido el desarrollo de computadores paralelos viables comercialmente con decenas, cientos y hasta miles de microprocesadores. Un computador paralelo es un conjunto de procesadores capaces de cooperar en la solución de un problema. Esta definición incluye supercomputadores con cientos de procesadores, máquinas con múltiples procesadores, redes de estaciones de trabajo y redes de PCs (Clusters de PCs).

⋆

Email addresses: vksotomayor@utpl.edu.ec (Vanessa K. Sotomayor A.), jsvalle1@utpl.edu.ec (Jairo S. Valle R.), jjvillavicencio@utpl.edu.ec (John J. Villavicencio S.).

2. Estado del Arte:

2.1. Grid Computing:

Grid se define como un sistema de computación distribuido que permite compartir y gestionar información mediante la colaboración de diferentes nodos, estos nodos de computadoras con diferente hardware y software, las cuales se utilizan para procesar una tarea que demanda una gran cantidad de recursos y poder de procesamiento.[3]

Para el uso de un grid es necesario un software denominado middelware, que asegura la comunicación transparente entre diferentes dispositivos o nodos a ser utilizados

Entre sus ventajas principales tenemos:

- **Gran poder de procesamiento:** Al combinar el poder de varias computadoras se puede procesar más datos.
- **Aprovechamiento de recursos existentes:** Se puede usar computadoras en los momentos en que no están siendo utilizadas para que ayuden con el procesamiento de datos en la grid.
- **No hay límite por espacio físico:** Los equipos que forman la grid pueden estar a gran distancia entre sí. Incluso pueden estar en diferentes continentes.
- **Brinda flexibilidad ante posibles fallos:** Si una máquina que forma parte del grid falla, el sistema lo

reconoce y envía los datos no procesados a otra máquina.
- **Ahorro en costos:** Cuando se necesita una gran capacidad de procesamiento, incurrir en gastos de equipos con ese poder no es una buena idea para medianas y pequeñas compañías por lo que una Grid es una opción viable y considerablemente más barata.

2.2. Middleware (Apache Spark):

El middleware seleccionado para la implementación del cluster es Apache Spark, es una herramienta con un gran potencial, actualmente es muy utilizada en Big Data y Machine Learning. Apache Spark combina un sistema de computación distribuida a través de clusters con ordenadores de una manera sencilla y elegante para escribir programas. Un cluster es un sistema basado en la unión de varios servidores que trabajarán de forma paralela como si se tratara de uno solo, es decir, tenemos un servidor formado por otros servidores. Podemos tener un cluster de 2, 10, 20 ó 100 servidores. Cuantos más servidores tengamos, más potencia de hardware estará disponible. Gracias a que Apache Spark hace un uso eficiente de las máquinas que formen un cluster, el sistema será escalable y realizará un correcto balanceo de carga.

2.3. Arquitectura de un cluster de Apache Spark:

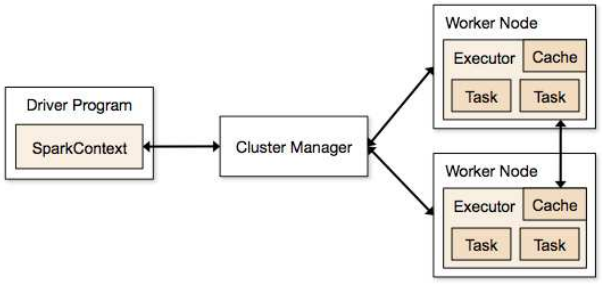


Figura 1. Arquitectura Apache Spark

En la arquitectura de un cluster montado con Apache Spark se puede observar que existen distintos agentes Fig.1. El primero de ellos es el controlador. Tras esto, encontramos el cluster manager y finalmente los workers. De una manera muy resumida, el manager hace las veces de maestro y el worker de esclavo. Conviene conocer algunos términos:[4]

- **Aplicación:** programa desarrollado sobre Spark por un programador.
- **Controlador del programa (driver program):** es el proceso que ejecuta la función main de la aplicación.
- **Cluster manager:** servicio externo utilizado para gestionar los recursos de los workers.
- **Worker:** nodo (servidor) que puede ejecutar código Spark.
- **Task:** operaciones (tareas) a realizar por un worker.

2.4. Modelo de programación:

Un programa típico se organiza de la siguiente manera[1]:

a) A partir de una variable de entorno llamada context se crea un objeto RDD leyendo datos de fichero, bases de datos o cualquier otra fuente de información.
b) Una vez creado el RDD inicial se realizan transformaciones para crear más objetos RDD a partir del primero. Dichas transformaciones se expresan en términos de programación funcional y no eliminan el RDD original, sino que crean uno nuevo.
c) Tras realizar las acciones y transformaciones necesarias sobre los datos, los objetos RDD deben converger para crear el RDD final. Este RDD puede ser almacenado.

Un pequeño ejemplo de código en Python que cuenta el número de palabras que contiene un archivo sería el siguiente:

```
1 my_RDD = spark.textFile("hdfs://...")
2 words = my_RDD.flatMap(lambda line : line.
      split(" "))
3       .map(lambda word : (word, 1))
4       .reduceByKey(lambda a, b : a+b)
5 words.saveAsTextFile("hdfs://...")
```

Spark ofrece más de 80 operadores de alto nivel que hacen que sea fácil de construir aplicaciones paralelas. Y se puede utilizar de forma interactiva desde las conchas Scala, Python y R.

3. Implementación del Cluster

3.1. Estudio a nivel de Hardware, Software y Red

3.1.1. Descripción de Hardware

Modelo	Porcesador	Memoria	Ram	Tarjeta de red	Propietario	Rol
Toshiba C45-ASP4311FL	Core i5 3230M (2.60 GHz)	750 GB	6 GB	Ethernet 10/100 LAN	Vanessa Sotomayor	Manger
Dell Inspiron 15r - N5110	2.ª generación de Intel® Core i7-2630QM (2,0 GHz, 6 MB, 4 N)	750 GB	DDR3 SDRAM de 8.192 MB (8GB)	Ethernet 10/100 LAN	John Villavicencio	Worker
Dell Inspiron M501R	Core i5	750 GB	DDR3 SDRAM de 4 GB	Ethernet 10/100 LAN	Jairo Valle	Worker

Figura 2. Descripción de hardware.

3.1.2. Descripción de Software

- **Sistema Operativo:** Se ha elegido usar Ubuntu 16.04 o distribuciones basadas en Ubuntu para la instalación y configuración del middleware Apache Spark.

- **Herramientas de monitoreo:**
  Apache Spark cuenta con un panel de administración en el cual se puede monitorizar como se encuentra funcionando la conexión del cluster y a la vez obtener datos sobre la ejecución de programas en el cluster Fig3.

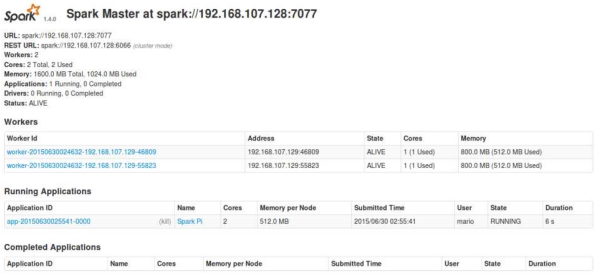


Figura 3. Monitoreo Apache Sapark.

3.1.3. Descripción de Red

Para nuestro proyecto trabajaremos con la topología de red en estrella, la misma que reduce la posibilidad de fallo de red conectando todos los nodos a un nodo central. Utilizaremos el protocolo TCP/IP que es un conjunto de protocolos estándar para conectar equipo y crear redes. El servicio de Cluster Server requiere el conjunto de protocolos TCP/IP para las comunicaciones internas y con los clientes. Fig.4 TCP/IP permite establecer comunicación en un entorno heterogéneo, es muy adecuado como protocolo de interconexión de redes corporativas, debido a que proporciona interoperabilidad entre diferentes tipos de sistemas informáticos.

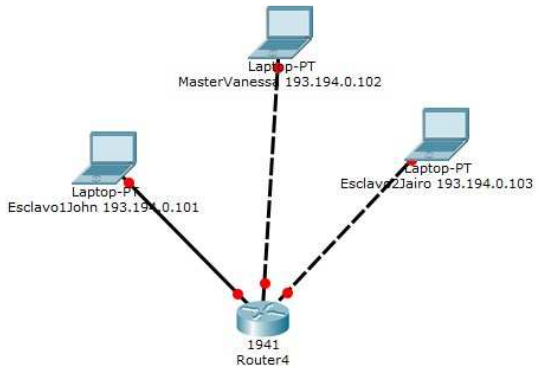


Figura 4. Topología de Red.

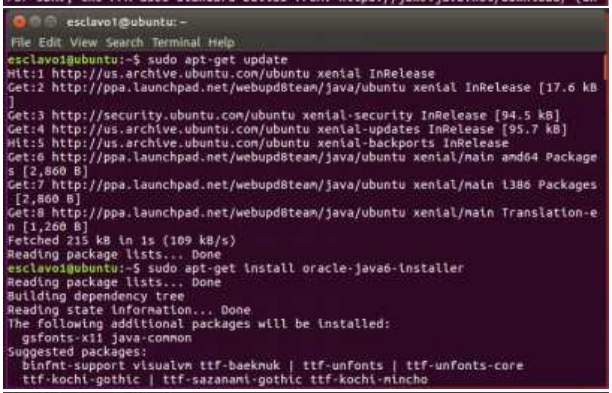
3.2. Implementación de la herramienta

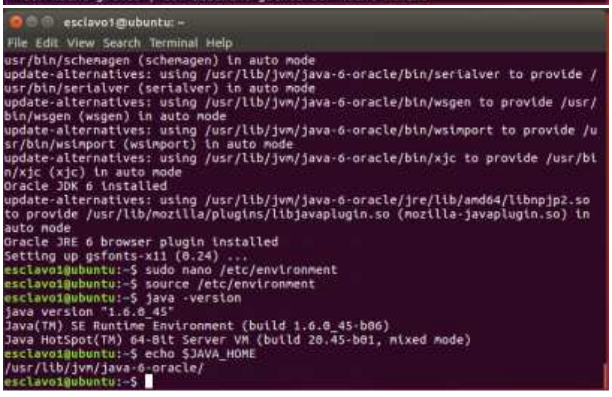
3.2.1. Requerimientos de instalación

a) **Java,** se puede utilizar tanto la versión 6 como la 8.
b) **Scala,** cualquier versión compatible.
c) Acceso remoto por SSH entre máquinas.
d) Apache Spark pre-compilado para Hadoop.
e) Mínimo dos máquinas virtuales, con el mismo sistema operativo y mismos nombres de usuario

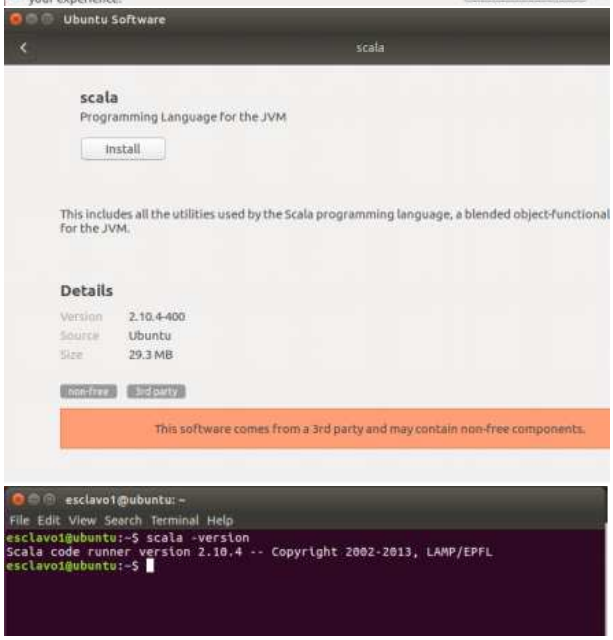
3.2.2. Instalación de la herramienta

3.2.2.1. Instalación y configuración de Java6

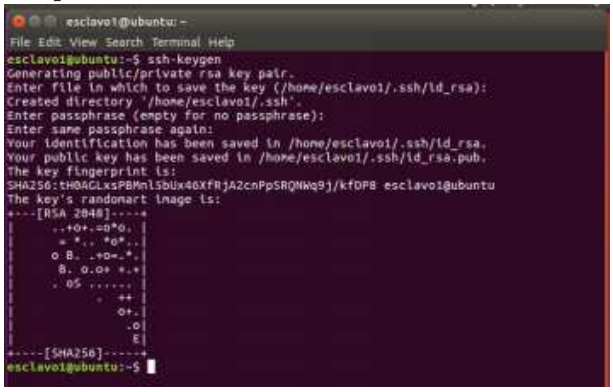


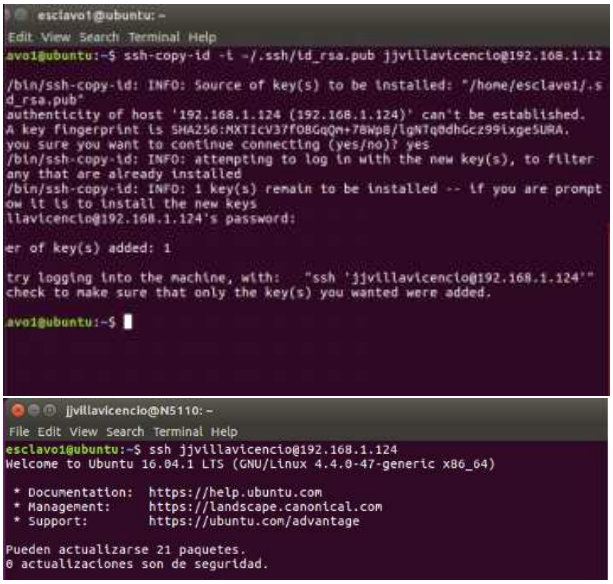


3.2.2.2. Instalación Scala 2.10 .

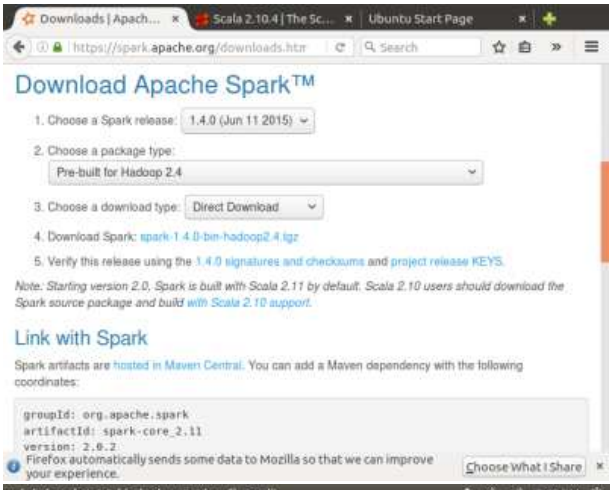


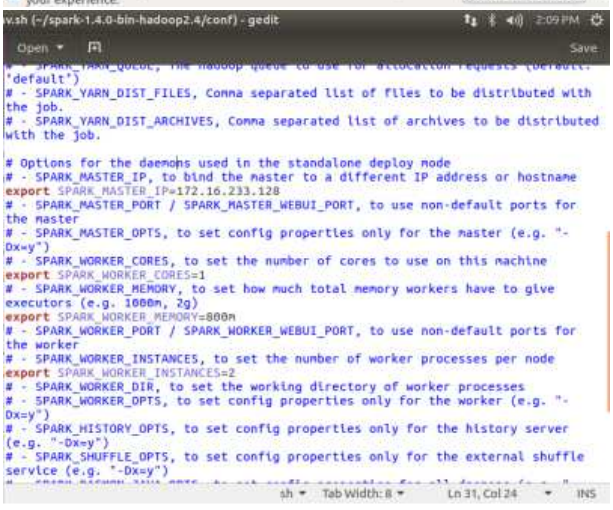
3.2.2.3. Configuración de acceso SSH entre máquinas .

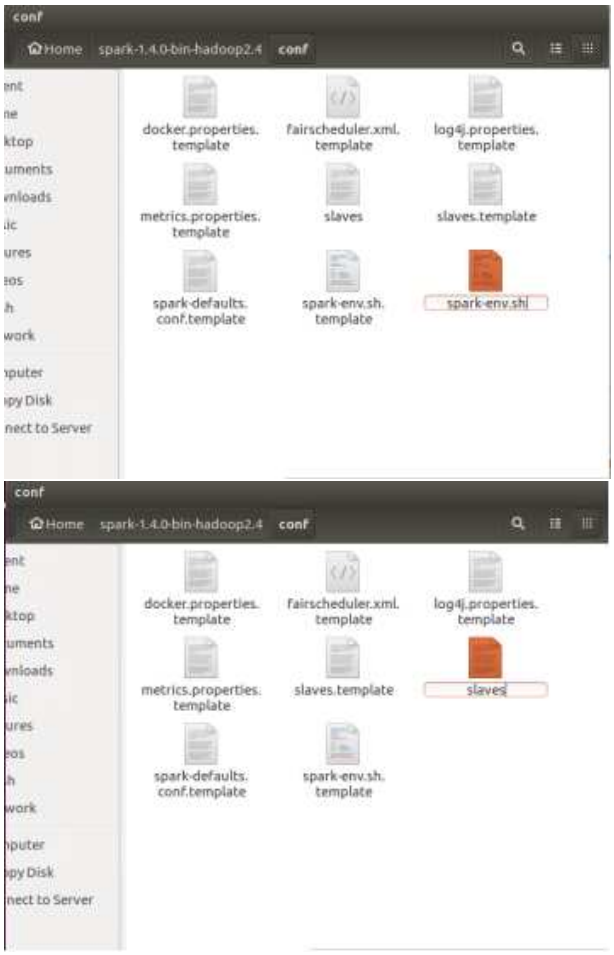


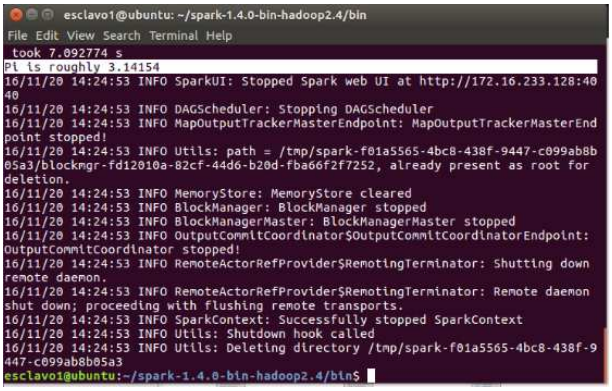


3.2.2.4. Instalación de Apache Spark 1.4.0 .



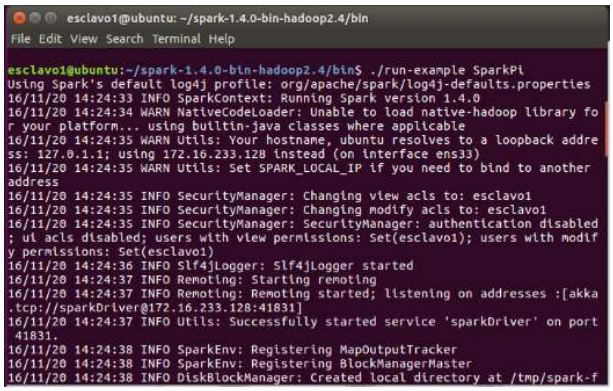




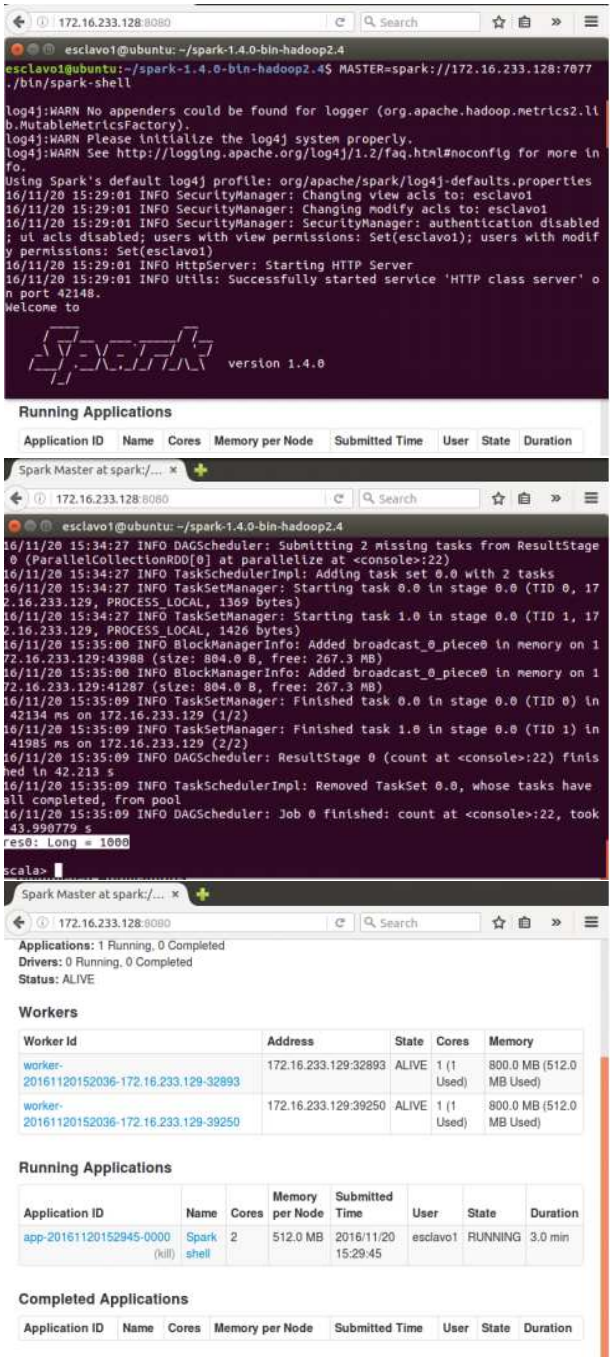


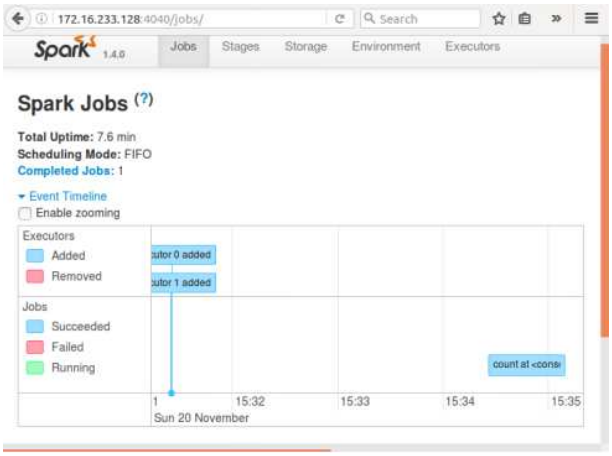
3.2.3. Pruebas de instalación

3.2.3.1. Prueba en nodo máster .



3.2.3.2. Prueba en cluster .





4. Implementación de la Aplicación Paralela

4.1. Tema, Problemática y descripción funcionamiento de la Aplicación

Procesamiento de un dataset, el objetivo de esta aplicación es dar una carga de trabajo que exija al cluster, lo que se logrará procesando un archivo de datos y clasificando estos a través de determinados parámetros.[2]

4.2. Algoritmo Secuencial

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """Procesar DataSet en secuencial"""
4
5  from time import time
6  from collections import Counter
7
8  __tiempo_inicial__ = time()
9
10 __dataSet__ = open('u.user', 'r')
11 __lineas__ = __dataSet__.readlines()
12
13 def intervalo_edad(edad):
14     """Intervalos de edad"""
15     if edad < 10:
16         return '0-10'
17     elif edad < 20:
18         return '10-20'
19     elif edad < 30:
20         return '20-30'
21     elif edad < 40:
22         return '30-40'
23     elif edad < 50:
24         return '40-50'
25     elif edad < 60:
26         return '50-60'
27     elif edad < 70:
28         return '60-70'
29     elif edad < 80:
30         return '70-80'
31     else:
32         return '80+'
```

```
33
34      def conversor_edad(data):
35          """pasar edad a un intervalo de
        edad"""
36          datos_conv = []
37          for linea in data:
38              datos_conv.append(linea.split("
        |"))
39
40          for usuario in datos_conv:
41              usuario[1] = intervalo_edad(int
        (usuario[1]))
42          return datos_conv
43
44      def usuarios_coinciden(intervalo, data)
        :
45          """Cantidad e usuarios que
        coinciden con el intervalo de edad"""
46          coincidencias = []
47          for usuario in data:
48              if usuario[1] == intervalo:
49                  coincidencias.append(
        usuario)
50          return coincidencias
51
52      def usuarios_profesion(data):
53          """Clasificar usuarios por
        profesion"""
54          profesiones = []
55          for usuario in data:
56              profesiones.append(usuario[3])
57          print Counter(profesiones)
58          return Counter(profesiones)
59
60      def extremos(data):
61          """Obtener cantidad de edades
        marginales"""
62          extremos_dic = {}
63          extremos_dic['pasados_edad'] = 0
64          extremos_dic['menor_edad'] = 0
65          for usuario in data:
66              if usuario[1] == "70-80":
67                  extremos_dic['pasados_edad'
        ] += 1
68              if usuario[1] == "0-10":
69                  extremos_dic['menor_edad']
        += 1
70          return extremos_dic
71
72      __cambiar_edades__ = conversor_edad(
        __lineas__)
73      __usuario_filtrados__ =
        usuarios_coinciden("20-30",
        __cambiar_edades__)
74      __edade_extremos__ = extremos(
        __cambiar_edades__)
75
76      print "El total de usuarios es: ", len(
        __lineas__)
77      print "Total de usuarios por profesion:
         ", usuarios_profesion(
        __usuario_filtrados__)
78      print "Usuarios entre 0-10: ",
        __edade_extremos__['menor_edad']
79      print "Usuarios entre 70-80: ",
        __edade_extremos__['pasados_edad']
80
81      __tiempo_final__ = time()
82      __tiempo_ejecucion__ = __tiempo_final__
        - __tiempo_inicial__
83
84      print 'El tiempo de ejecucion fue:',
        __tiempo_ejecucion__ #En segundos
85
```

4.3. *Algoritmo Paralelo*

```
1       from pyspark import SparkContext,
        SparkConf
2
3       conf = SparkConf().setAppName('
        MyFirstStandaloneApp')
4       sc = SparkContext(conf=conf)
5
6       userRDD = sc.textFile("/usr/lib/
        data_backup/opt/spark_usecases/movie/ml
        -100k/u.user")
7
8       def parse_N_calculate_age(data):
9           userid, age, gender, occupation,
        zip = data.split("|")
10          return  userid, age_group(int(age))
        , gender, occupation, zip, int(age)
11
12      def  age_group(age):
13          if age < 10:
14              return '0-10'
15          elif age < 20:
16              return '10-20'
17          elif age < 30:
18              return '20-30'
19          elif age < 40:
20              return '30-40'
21          elif age < 50:
22              return '40-50'
23          elif age < 60:
24              return '50-60'
25          elif age < 70:
26              return '60-70'
27          elif age < 80:
28              return '70-80'
29          else:
30              return '80+'
31
32      data_with_age_bucket = userRDD.map(
        parse_N_calculate_age)
33
34      RDD_20_30 = data_with_age_bucket.filter
        (lambda line: '20-30' in line)
35
36      freq = RDD_20_30.map(lambda line: line
        [3]).countByValue()
37
38      print "El total de usuarios es: ",
        userRDD.count()
39
```

```
    print "Total de usuarios por profesion
    ", dict(freq)

    Under_age = sc.accumulator(0)
    Over_age = sc.accumulator(0)

    def outliers(data):
        global Over_age, Under_age
        age_grp = data[1]
        if age_grp == "70-80":
            Over_age += 1
        if age_grp == "0-10":
            Under_age += 1
        return data

    df = data_with_age_bucket.map(outliers)
    .collect()

    print "Usuarios entre 0-10: ",
    Under_age
    print "Usuarios entre 70-80: ",
    Over_age
```

4.3.1. *Descripción detallada de las funciones y librerías paralela que se utiliza*

Acerca del dataset

U.user: Información demográfica de usuarios, separada por barras verticales con los siguientes campos, user id — age — gender — occupation — zip code

Ejemplo:

```
    1|24|M|technician|85711
    2|53|F|other|94043
    3|23|M|writer|32067
    4|24|M|technician|43537
    5|33|F|other|15213
    6|42|M|executive|98101
    7|57|M|administrator|91344
    8|36|M|administrator|05201
    9|29|M|student|01002
```

Enlace para descargar dataset:
http://grouplens.org/datasets/movielens/

Leer datos con Spark RDD (Resilient Distributed Dataset)

```
    userRDD = sc.textFile("/usr/lib/
    data_backup/opt/spark_usecases/movie/ml
    -100k/u.user")
```

Contar numero de registros

```
    userRDD.count()
```

Vamos a crear una función definida por el usuario para dividir a los usuarios en grupos de edad:

```
def parse_N_calculate_age(data):
    userid, age, gender, occupation, zip =
    data.split("|")
    return  userid, age_group(int(age)),
    gender, occupation, zip, int(age)
def  age_group(age):
    if age < 10:
        return '0-10'
    elif age < 20:
        return '10-20'
    elif age < 30:
        return '20-30'
    elif age < 40:
        return '30-40'
    elif age < 50:
        return '40-50'
    elif age < 60:
        return '50-60'
    elif age < 70:
        return '60-70'
    elif age < 80:
        return '70-80'
    else:
        return '80+'
```

```
data_with_age_bucket = userRDD.map(
    parse_N_calculate_age)
```

Ahora, vamos a analizar el grupo de edad "20-30"para su posterior análisis

```
    RDD_20_30 = data_with_age_bucket.filter
    (lambda line: '20-30' in line)
```

Vamos a contar el número de usuarios por su profesión en el age_group determinado 2030

```
    freq = RDD_20_30.map(lambda line: line
    [3]).countByValue()
```

Ahora, usaremos Acumuladores para la detección de valores aleatorios en el conjunto de datos de película anterior. Supongamos que cualquier persona que caiga en el grupo de edad 70-80 sea marcada como over_age y cualquier persona que caiga en el grupo de edad 0-10 es marcado como under_age.

```
Under_age = sc.accumulator(0)
Over_age = sc.accumulator(0)

```

```
def outliers(data):
    global Over_age, Under_age
    age_grp = data[1]
    if age_grp == "70-80":
        Over_age += 1
    if age_grp == "0-10":
        Under_age += 1
    return data

```

4.3.2. Descripción detallada del funcionamiento de la aplicación

Básicamente la aplicación realizara tres operaciones con un dataset de usuarios:

- **Clasificación de usuarios por edad:**
  en el dataset a utilizar cada usuario tiene su edad con la intención de captar y procesar la mayor cantidad de usuarios se los agrupara en intervalos de edad por ejemplo de 20-30
- **Usuarios por profesión:**
  Una ves que tenemos los usuarios que coinciden con el intervalo de edad especificado (20-30), contaremos cuantos usuarios pertenecen a determinadas profesiones que se encuentran registras en el dataset.
- **Edades menores y mayores:**
  El ultimo calculo a realizar es contar la cantidad de usuarios que tienen edad máxima (70-80) y edad mínima(0-10), esto se lo hará de la totalidad del dataset.

5. Integración del Cluster y la Aplicación

5.1. Plan de Pruebas

El objetivo del plan de pruebas a realizar es comprobar la optimización de recursos cuando se trabaja con algoritmos paralelos bajo una arquitectura grid, en comparación con la versión secuencial desarrollada de igual manera dentro del proyecto

Pruebas de Conexión: Para verificar la conectividad se ha elaborado el siguiente proceso:

- Creación de llaves publicas SSH en los nodos worker para que el nodo master pueda acceder sin ingresar contraseñas.
- Levantar el servicio del cluster /sbin/start-all.sh que conecta todos los nodos configurados
- Ejecutar la interfaz web y constatar los nodos worker.

Pruebas de Ejecución: Las pruebas de ejecución se realizan tanto para la versión secuencial del algoritmo como para la paralela.

- En la versión secuencial del programa, se trabajará con el archivo data_secuencial.py, en el cual el algoritmo esta programado con funciones normales.
- En la versión paralela se trabajará con el archivo data_paralelo.py en el cual el algoritmo esta programado con la librería pyspark exclusiva para el cluster.
- La data a utilizar se encuentra en el archivo u.user con variaciones de 19803, 39606 y 60292 datos, el mismo que se encontrará disponible en cada nodo

5.2. Comparación mediante los criterios de evaluación de las versión secuencial y paralela

Tiempos de respuesta:

DATOS	SECUENCIAL	PARALELO
19803	0.03461	29.360435
39606	0.07382	19.508621
60292	0.10606	17.20789

Coste secuencial:
$Tej * \#Proc = 0{,}64347 * 5 = 3{,}2173$

Coste paralelo:
$Tej * \#Proc = 22{,}02564 * 5 = 110{,}12824$

Tiempo de rendimiento Secuencial:
$Tf–Ti = 0{,}10606–0{,}03461 = 0{,}07145$

Tiempo de rendimiento Paralelo:
$Tf–Ti = 17{,}20789–29{,}360435 = 12{,}15254$

Aceleración:
$Ts/Tp = 0{,}07145/12{,}15254 = 0{,}005879$

Eficiencia:
$Ts/p(Tp) * 100 = 0{,}07145/5(12, 15254) = 0{,}117\,\%$

Valor de ejecución:
$0 <= Sp < n\ 0 <= 0{,}005879 < 110, 1282$

Eficiencia:
$0 <= tf < 1\ 0 <= 0{,}117\,\% < 1$

5.3. *Gráficas de la comparación de las dos versiones*

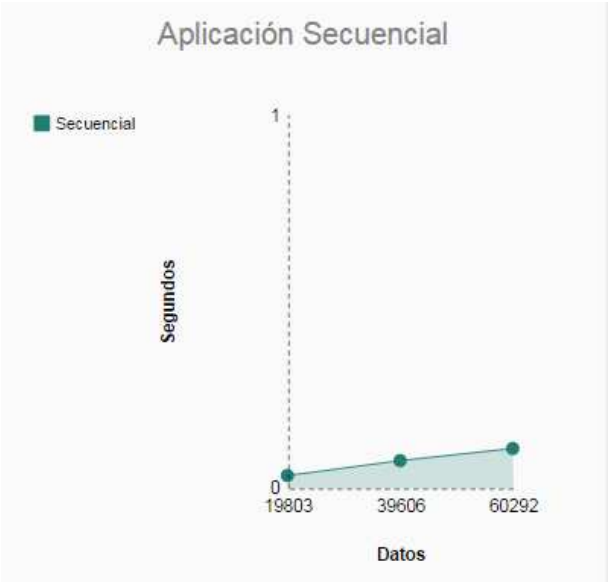


Figura 5. Tiempos respuesta secuencial.

Figura 6. Tiempos respuesta paralelo.

5.4. *Análisis del rendimiento*

Una vez obtenidos los tiempos de ejecución de la aplicación tanto en paralelo como en secuencial, nos podemos dar cuenta primero, que para trabajar con Apache Spark se debe usar grandes volúmenes de datos debido que esta orientado a BigData, segundo que con la cantidad de datos que se realizó las pruebas la ejecución secuencial obtuvo mejores tiempos que la ejecución paralela, y tercero que se puede notar que mientras aumenta el volumen de datos a procesar Apache Spark va reduciendo su tiempo de respuesta muy contrario a la ejecución en secuencial que va aumentado sus tiempos de respuesta.

6. Conclusiones y Recomendaciones

- Spark es un entorno con múltiples funcionalidades, que simplifica enormemente la gestión de grandes cantidades de datos de forma transparente. La magia de Spark reside en que el programa escrito en el ejemplo, puede ejecutarse tanto en un clúster local, como en un clúster distribuido compuesto por multitud de nodos sin cambiar ni una sola línea de código.
- Apache Spark es una herramienta útil y eficiente para tareas de procesamiento masivo de datos. Está en constante desarrollo y se actualiza frecuentemente. Además, su documentación es muy completa y la comunidad cada vez se hace más grande.
- Se recomienda usar Apache Spark para trabajar con procesamiento de DataSets o grandes volúmenes de información, para aprovechar el cluster.
- Se recomienda familiarizarse con la librería pyspark para realizar programas completamente compatibles con Spark desarrollados en Python.

Referencias

[1] Mario Pérez Esteso. Apache spark: qué es y cómo funciona, sep 2015.
[2] https://www.dezyre.com/. Pyspark tutorial-learn to use apache spark with python, dec 2015.
[3] Luis Mora. ¿qué es la computación grid?, aug 2008.
[4] Francisco Javier Pulido. Spark (iii) - ¿cómo crear y configurar un clúster?, may 2014.