UOC

Trabajo de Fin de Grado

Estudio de un entorno de computación distribuida con Hadoop

Alumno: Francisco J. Alvarez Goikoetxea
Profesor: Francesc Guim Bernat

A Itzi
Porque el camino no se puede andar solo.

A Charo y Eduardo
Porque alguien te tiene que enseñar el valor de andar los caminos, y mostrarte el sacrificio que requiere.

Indice

Capítulo 0 – Presentación del Trabajo de Fin de Grado 6

0.1 – Índice 7

0.2 – Motivación del proyecto 8

0.3 – Objetivos 10

0.4 – Requerimientos 11

0.5 – Fases del proyecto 12

0.6 – Planificación del proyecto 16

0.6.1 – Calendario 16

0.6.2 – Diagrama de Gantt 17

Capítulo 1 – Evaluación del rendimiento 18

1.1 – Índice del capítulo 1 19

1.2 – Introducción 20

1.3 – Linpack 21

1.4 – Necesidad de un nuevo modelo de evaluación 22

1.5 – NPB 23

1.6 – Conclusiones 24

1.7 – Bibliografía 25

Capítulo 2 – Construcción de la aplicación de tratamiento de logs para MapReduce 26

2.1 – Índice del capítulo 2 27

2.2 – Introducción: ¿Qué es Hadoop? 28

2.3 – Hadoop 1: MapReduce 29

2.3.1 – Estructura de un programa para MapReduce 30

2.3.1.1 – La clase Mapper 30

2.3.1.2 – La clase Reducer 30

2.3.1.3 – La clase Driver 30

2.4 – Hadoop 2: YARN & MapReduce 2 31

2.4.1 – Arquitectura de una ejecución sobre YARN 32

2.4.2 – YARN versus MapReduce 1 33

2.5 – Selección de archivos para la aplicación y funcionalidad de la misma 33

2.6 – Construcción de la aplicación. Comentarios al código 34

2.7 – Pruebas en modo local y pseudo distribuido 37

2.8 – Conclusiones 40

2.9 – Bibliografía 41

Capítulo 3 - Construcción de la aplicación de tratamiento de logs para Spark 42

3.1 – Índice del capítulo 3 43

3.2 – Introducción. ¿Por qué Spark? 44

3.3 – Modelo de programación de Spark 44

3.3.1 – RDDs. 44

3.3.2 – Spark es “vago” 45

3.3.3 – Arquitectura de Spark 45

3.4 – Construcción de la aplicación. Comentarios al código 46

3.5 – Prueba de la aplicación 47

3.6 – Conclusiones 48

3.7 – Bibliografía 49

Capítulo 4 - Construcción de la aplicación de cálculo iterativo para MapReduce 50

4.1 – Índice del capítulo 4 51

4.2 – Introducción 52

4.3 – Selección del algoritmo a implementar 52

4.4 – Construcción de la aplicación. Comentarios al código 53

4.5 – Prueba de la aplicación 55

4.6 – Conclusiones 55

Capítulo 5 - Construcción de la aplicación de cálculo iterativo para Spark 57

5.1 – Índice del capítulo 5 58

5.2 – Introducción 59

5.3 – Construcción de la aplicación. Comentarios al código 59

5.4 – Prueba de la aplicación 61

5.5 – Conclusiones 62

Capítulo 6 – Generadores de tráfico 63

6.1 – Índice del capítulo 6 64

6.2 – Introducción 65

6.3 – Generadores de tráfico basados en HadoopLog y SparkLog 69

6.4 – Generador de tráfico genérico 71

6.5 – Conclusiones 73

Capítulo 7 – Pruebas experimentales 75

7.1 – Índice del capítulo 7 76

7.2 – Introducción 77

7.3 – HadoopLog en Hadoop 1 y Hadoop 2 78

7.4 – HadoopIter en Hadoop 2 81

7.5 – SparkLog y SparkIter sobre SPARK4MN 83

7.6 – Conclusiones 87

Capítulo 8 – Conclusiones sobre el TFG 88

Anexo A – Instalación del entorno de trabajo 92

Anexo B – Notas sobre el código adjunto al proyecto 100

Capítulo 0

Presentación del Trabajo de Fin de Grado

0.1 - Indice

0.1 – Índice

0.2 – Motivación del proyecto

0.3 – Objetivos

0.4 – Requerimientos

0.5 – Fases del proyecto

0.6 – Planificación del proyecto

0.6.1 – Calendario

0.6.2 – Diagrama de Gantt

0.2 - Motivación del proyecto

Sistemas distribuidos y supercomputación. Dos conceptos que para el usuario de a pie podrían parecer inconexos por un lado y lejanos, e incluso etéreos, por otro. Nada más lejos de la realidad.

Si nos centramos en los sistemas distribuidos, todos los días interactuamos con muchos de ellos, como son las diferentes redes sociales, como Facebook o Instagram, los buscadores como Google o Bing, o los omnipresentes programas de mensajería instantánea, como Whatsapp o Telegram, por no hablar de los sistemas informáticos existentes en muchas grandes empresas. Estamos tan acostumbrados a estos servicios, nos parecen algo tan sumamente dado por hecho, que no nos damos cuenta de toda la tecnología que los soporta. Numerosas granjas de servidores, repartidas por todo el mundo, se encargan de que dispongamos de estos servicios allá donde vayamos. Estas granjas no son sino grandes estructuras que contienen un sistema distribuido de varios miles de nodos (los centros de datos de Google llegan a 15.000), que nos permiten acceder a nuestras aplicaciones favoritas, garantizándonos la movilidad y la conectividad.

Si pensamos en la supercomputación, tendemos a evocar los llamados superordenadores. Máquinas específicamente diseñadas para realizar billones de operaciones por segundo, como pueden ser el Tianhe-2 o la serie XC de Cray. Las necesidades computacionales han ido creciendo a lo largo de los años, tanto a nivel científico y profesional, como a nivel del usuario doméstico. Un “gamer”, si se me permite la palabra, posee un equipo que hace poco más de dos décadas hubiese estado en las primeros puestos de la lista de ordenadores más potentes del mundo. Los actuales estudios de animación y edición de video poseen equipos con unas capacidades de cómputo muy elevadas, y el mundo científico pide un poder computacional cada vez más elevado para poder realizar modelos complejos que puedan arrojar luz sobre problemas con un gran número de variables, como la meteorología y el cambio climático, el movimiento magmático en el interior de la Tierra o la formación del universo.

El mundo de la supercomputación, o HPC de sus siglas en inglés de High Performance Computing, ha vivido no ya una transformación, sino un verdadero cambio de paradigma. En el mundo se generan diariamente ingentes cantidades de datos que necesitan ser almacenados y, posteriormente, analizados para extraer conocimiento de ellos. No precisamos sólo de una capacidad bruta de proceso, como en la supercomputación clásica, en donde ejecutábamos un programa sobre unos datos más o menos estáticos, sino que necesitamos almacenar y analizar ingentes cantidades de datos en tiempo real. Todo ello en un entorno cambiante, que debe adaptarse a nuevos modelos de programación y a diferentes arquitecturas continuamente.

Ya he mencionado la serie XC de Cray a modo de ejemplo de un computador con una elevada capacidad de cálculo. Sin embargo, si vamos a la página web de este fabricante, podemos ver que tiene una serie dedicada al almacenamiento a gran escala y otra dedicada al análisis de datos. Ambas series son relativamente recientes y muestran la necesidad de adaptación al nuevo entorno creado, entre otros, por el Big Data. Sin embargo, máquinas como los Cray aún son caros y con arquitecturas muy específicas, aplicables a problemas concretos. Las nuevas necesidades de sistemas cada vez más potentes han hecho que los fabricantes busquen equipos comerciales, mucho más baratos, empleados en sistemas que permitan escalar hasta cientos de miles de nodos. Como estos equipos comerciales tienen una mucho más alta probabilidad de fallar que los anteriores, se tienen que implementar formas de controlar los fallos y errores de los nodos individuales, para que el sistema como conjunto sea fiable.

Es aquí donde ambos mundos confluyen, donde los sistemas distribuidos son superordenadores por derecho propio (ciertamente, no tan optimizados como los sistemas HPC), y donde los superordenadores se han convertido en sistemas distribuidos de alta velocidad. Donde unos se han mirado en el espejo de los otros para adaptarse a las nuevas necesidades y dar soluciones a la demanda de nuevos modelos.

Es en este entorno dinámico y cambiante donde quiero desarrollar mi trabajo. Existen muy diversas ofertas para poder efectuar computación distribuida, desde las PaaS hasta sistemas que podemos instalar en nuestro clúster para poder llevar una gestión directa y sin depender de servicios externos.

Hadoop es un sistema de gestión de computación distribuida que está presente en muchos sistemas conocidos y que permite tanto el almacenamiento en un sistema de archivos distribuido, como la computación distribuida. Además, es un sistema en continuo cambio y desarrollo, por lo que lo he elegido para llevar a cabo este trabajo de fin de grado. Para muestra un botón; el desarrollo de Hadoop es tan rápido, que el libro "Learning Hadoop", 2ed, de Turkington y Modena, fue publicado en Febrero de 2015 y en Agosto del mismo año ya tuve problemas para seguir los ejemplos debido a los cambios en la versión más reciente de Hadoop.

No quiero terminar este apartado acerca de la motivación del proyecto sin mencionar una frase del libro "Hadoop in Practice", 2ed (Alex Holmes, editorial Manning):

"MapReduce and YARN are not for the faint of heart…"

¡Qué mejor forma de empezar un trabajo que ésta!

0.3 - Objetivos

El objetivo de este proyecto es efectuar un estudio de la evolución de Hadoop desde su aparición hasta hoy en día, tratando de ver y explicar las mejoras en su rendimiento.

Puesto que Hadoop es hoy en día un conjunto de aplicaciones integradas, con diferentes modelos de programación y aplicaciones, un estudio completo de todo el conjunto, que sus creadores llaman ecosistema, sería demasiado extenso para este trabajo. Es por ello que me centraré en dos aspectos del mismo:

1. MapReduce en Hadoop 1 y su evolución hacia una aplicación sobre YARN en Hadoop 2.
2. Spark, que es una optimización de código para poder ejecutar determinadas aplicaciones más rápidamente.

Para llevar a cabo este estudio desarrollaré e implementaré en las tres plataformas una aplicación de tratamiento de archivos de log generados por un servidor, así como una aplicación de cálculo numérico.
Ambas aplicaciones se ejecutarán para diferentes valores de N, siendo N en el caso de los archivos de log el número de líneas del archivo, y para el caso de cálculo numérico el número de iteraciones.
Las métricas a emplear para la comparación de las ejecuciones serán las proporcionadas por las plataformas al terminar las mismas.

0.4 - Requerimientos

En principio, Hadoop está pensado para ejecutarse sobre un clúster lo que complica el desarrollo de este trabajo. Afortunadamente, existe un modo de ejecución local que permite correr las aplicaciones en una sola máquina y dentro de una sola máquina virtual de Java. Existe, además, un modo de ejecución denominado pseudo distribuido, donde los diferentes hilos de ejecución se ejecutan en su propia máquina virtual de Java dentro de una misma máquina física. Estos son los modos recomendados para probar y depurar las aplicaciones que escribamos antes de lanzarlas en el clúster.

Hadoop puede ejecutarse en diferentes plataformas y, aunque existen modos de ejecutarlo sobre Windows, está desarrollado para sistemas *NIX. Por lo tanto, para la realización del trabajo emplearé una plataforma x86_64 con S.O. Linux.
Necesitaré dos versiones de Hadoop, que son la 1 y la 2. Aunque existen múltiples distribuciones, tanto comerciales como comunitarias, emplearé los binarios proporcionados por Apache, tanto por ser de código libre como por la gran cantidad de documentación existente.

Para la realización del trabajo voy a emplear tres máquinas; mi ordenador personal, la máquina de la UOC, donde dispongo de una cuenta, y el Superordenador del BSC Mare Nostrum (MN a partir de ahora), donde dispongo de otra cuenta. Hadoop precisa de un JDK en su versión 1.7 o superior. Puesto que tanto en la máquina de la UOC como en el MN está instalada la versión 1.7, será la que emplee por motivos de compatibilidad. Todas ellas son plataformas x86_64 sobre las que corre Linux.

Por último, en el desarrollo de aplicaciones Java para Spark podemos utilizar diversas herramientas. En mi caso emplearé Maven, puesto que es una herramienta de fácil acceso, código libre y sobradamente probada. Puesto que es una herramienta de construcción de la aplicación y, como he comentado, las aplicaciones se pueden probar en modo local, sólo será necesario tenerla instalada en mi máquina personal, donde procederé a desarrollar y probar las aplicaciones. El grueso de las ejecuciones se hará sobre el MN, puesto que en modo local dispongo de hasta 16 hilos de ejecución.

0.5 - Fases del proyecto

A continuación describiré, de forma breve, las diferentes fases de las que se compone este proyecto.

Creación del plan de trabajo

Del 19/09/2015 al 27/09/2015: 9 días.

El resultado final de esta fase es la creación del presente documento, y que se corresponde con la PEC1.

A pesar de que la siguiente fase del proyecto se corresponde con la recopilación de la documentación necesaria para poder llevar a cabo el trabajo, ha sido necesario documentarse previamente para hacerse una idea del funcionamiento las tecnologías involucradas, así como de la complejidad del proceso de desarrollo de las herramientas que se van a emplear para el estudio, objeto del presente proyecto. Gracias a este proceso de documentación previa, ha sido posible la creación de este documento.

Recopilación de documentación

Del 28/09/2015 al 04/10/2015: 7 días.

Esta fase va a requerir un trabajo tanto extensivo como intensivo. Extensivo en el sentido de que va a ser necesario recopilar una gran cantidad de información, e intensivo puesto que va a requerir una gran concentración y capacidad selectiva para decidir con qué material se va a trabajar finalmente.

Durante esta fase se va a proceder, de forma paralela, a consultar este material para ir adquiriendo los conocimientos necesarios para poder ejecutar las siguientes fases del proyecto. De todas maneras, el material se consultará de forma superficial en esta fase, siendo importante el proceso de selección, puesto que se consultará a un nivel mucho más exhaustivo en las fases posteriores del trabajo.

Instalación y puesta en marcha del entorno de trabajo

Del 05/10/2015 al 09/10/2015: 5 días.

Durante esta fase del proyecto deberemos instalar las diferentes plataformas con las que vamos a trabajar. Asimismo, se deberá probar que las tecnologías instaladas funcionan correctamente, ejecutando algunos de los ejemplos que dichas tecnologías traen por defecto.

Las plataformas que se van a emplear son:

- Hadoop 1.
- Hadoop 2.
- Spark.

Además, deberemos tener instalado un JDK, que en nuestro caso será el 1.7 por las razones explicadas en el apartado de requerimientos.

En el MN ya disponemos de los módulos de Java 1.7, Hadoop 2 y Spark, por lo que tan solo será necesario copiar los binarios de Hadoop 1.

En mi máquina personal deberé instalar y probar Maven.

Desarrollo de la aplicación de tratamiento de logs sobre MapReduce

Del 10/10/2015 al 22/10/2015: 13 días.

Como se ha mencionado anteriormente, durante esta fase se procederá a consultar, de forma exhaustiva, la documentación relativa a MapReduce, la descarga de archivos de registro de acceso público, teniendo cuidado en que sean lo suficientemente extensos para que los resultados de las pruebas sean significativos, y la construcción de la aplicación de tratamiento de logs sobre MapReduce.

Evidentemente, se efectuarán y evidenciarán pruebas del correcto funcionamiento de la misma.

En función de los archivos de acceso público que se obtengan, se decidirá la funcionalidad concreta de la aplicación que, en ningún caso, afectará al resultado del trabajo.

Implementación de la aplicación de tratamiento de logs sobre Spark

Del 23/10/2015 al 06/11/2015: 15 días.

Aquí se consultará la documentación relativa a Spark y se procederá a construir una aplicación equivalente a la de la fase anterior, pero que corra sobre este entorno. Como en el apartado anterior, se llevarán a cabo pruebas del correcto funcionamiento de la misma. No es necesaria la descarga de archivos puesto que, como es lógico, se emplearán los mismos que en el apartado anterior.

Pruebas de la aplicación de tratamiento de logs sobre Hadoop 1, Hadoop 2 y Spark

Del 07/11/2015 al 14/11/2015: 8 días.

Se llevarán a cabo ejecuciones de las aplicaciones desarrolladas en los dos apartados anteriores, con diferentes valores del número de líneas de los archivos de registro que se empleen como entrada. Asimismo, se procederá a la recopilación de las diferentes métricas para poder efectuar las comparaciones objeto del trabajo en la fase de análisis de datos.

Desarrollo de la aplicación de cálculo iterativo sobre MapReduce

Del 15/11/2015 al 01/12/2015: 17 días.

En esta fase, se desarrollará una aplicación de cálculo numérico para MapReduce. A pesar de que en la implementación de la aplicación de tratamiento de archivos de registro ya se habrá consultado exhaustivamente la documentación sobre MapReduce, la construcción de un programa que, en principio, no se ajusta a este modelo de programación requerirá de consultas adicionales.

Deberá decidirse, al comienzo de esta fase, la aplicación de cálculo numérico a implementar. Puesto que no es objeto de este trabajo el efectuar un estudio de las soluciones matemáticas de esta índole, se elegirá un método de resolución de ecuaciones de forma iterativa relativamente sencillo, como podría ser el método de Newton.

Como en los casos anteriores, se procederá a realizar pruebas del correcto funcionamiento de la aplicación.

Implementación de la aplicación de cálculo iterativo sobre Spark

Del 02/12/2015 al 18/12/2015: 17 días.

Se desarrollará una aplicación equivalente a la anterior para la plataforma Spark y se efectuarán pruebas de su correcto funcionamiento.

Pruebas de la aplicación de cálculo iterativo sobre Hadoop 1, Hadoop 2 y Spark.

Del 19/12/2015 al 28/12/2015: 10 días (el día de navidad festivo).

Se llevarán a cabo ejecuciones de las aplicaciones desarrolladas en los dos apartados anteriores. Se variará el número de iteraciones en la resolución del problema. Asimismo, se procederá a la recopilación de las diferentes métricas para poder efectuar las comparaciones objeto del trabajo en la fase de análisis de datos.

Análisis de los resultados

Del 29/12/2015 al 10/01/2016: 13 días (año nuevo y reyes festivos).

En este periodo, con las métricas recopiladas en las diferentes ejecuciones, se procederá a analizar y explicar los resultados obtenidos, relacionándolos con la evolución de Hadoop y las diferencias existentes entre las diferentes plataformas empleadas.

Se generarán gráficas que ayuden a explicar y comprender los resultados.

Creación de la memoria del trabajo

Del 11/01/2016 al 21/01/2016: 11 días.

En esta última fase, se procederá a la redacción de la memoria del TFG.

Entrega del trabajo

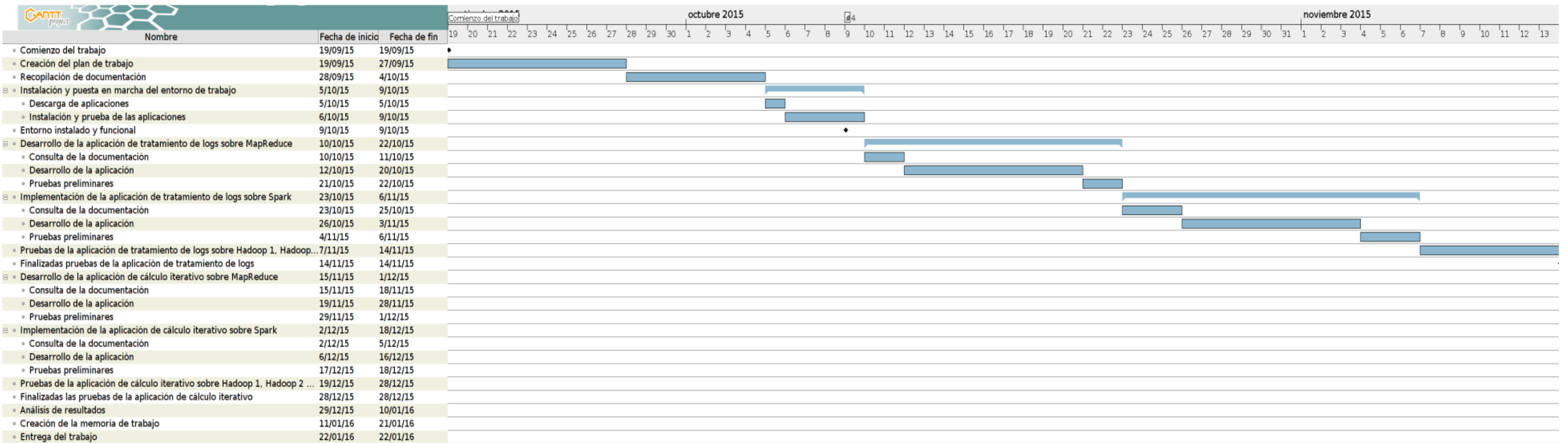
22/01/2016

0.6 - Planificación del proyecto

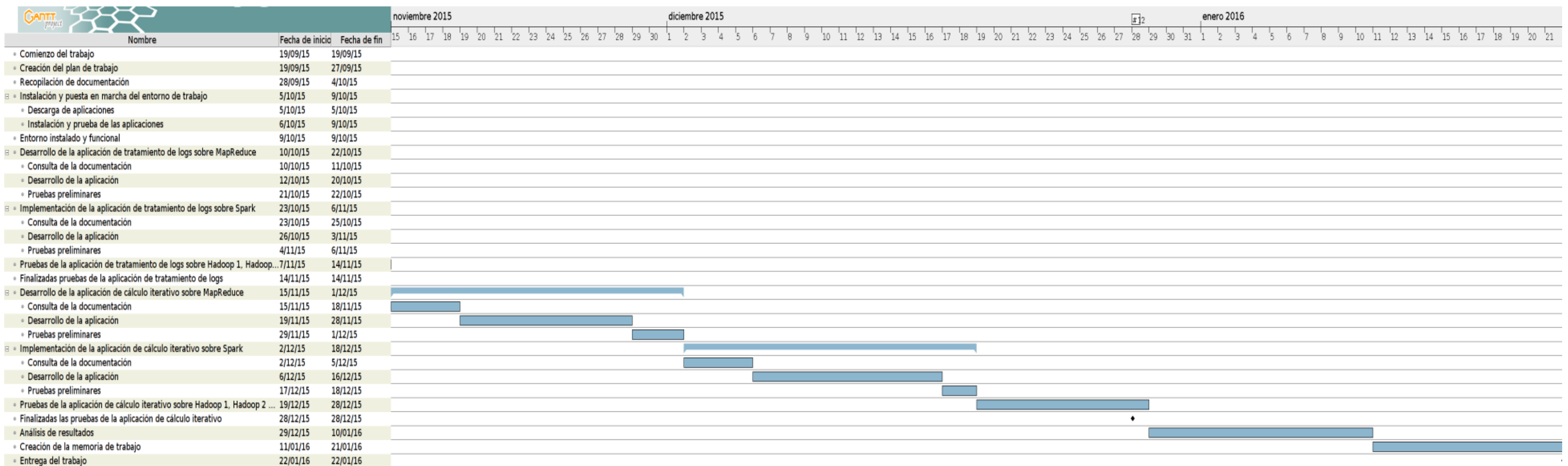
0.6.1 – Calendario del proyecto

Tarea	Inicio	Fin
Comienzo del trabajo	19/09/15	19/09/15
Creación del plan de trabajo	19/09/15	27/09/15
Recopilación de documentación	28/09/15	4/10/15
Instalación y puesta en marcha del entorno de trabajo	5/10/15	9/10/15
Descarga de aplicaciones	5/10/15	5/10/15
Instalación y prueba de las aplicaciones	6/10/15	9/10/15
Entorno instalado y funcional	9/10/15	9/10/15
Desarrollo de la aplicación de tratamiento de logs sobre MapReduce	10/10/15	22/10/15
Consulta de la documentación	10/10/15	11/10/15
Desarrollo de la aplicación	12/10/15	20/10/15
Pruebas preliminares	21/10/15	22/10/15
Implementación de la aplicación de tratamiento de logs sobre Spark	23/10/15	6/11/15
Consulta de la documentación	23/10/15	25/10/15
Desarrollo de la aplicación	26/10/15	3/11/15
Pruebas preliminares	4/11/15	6/11/15
Pruebas de la aplicación de tratamiento de logs sobre Hadoop 1, Hadoop...	7/11/15	14/11/15
Finalizadas pruebas de la aplicación de tratamiento de logs	14/11/15	14/11/15
Desarrollo de la aplicación de cálculo iterativo sobre MapReduce	15/11/15	1/12/15
Consulta de la documentación	15/11/15	18/11/15
Desarrollo de la aplicación	19/11/15	28/11/15
Pruebas preliminares	29/11/15	1/12/15
Implementación de la aplicación de cálculo iterativo sobre Spark	2/12/15	18/12/15
Consulta de la documentación	2/12/15	5/12/15
Desarrollo de la aplicación	6/12/15	16/12/15
Pruebas preliminares	17/12/15	18/12/15
Pruebas de la aplicación de cálculo iterativo sobre Hadoop 1, Hadoop 2 ...	19/12/15	28/12/15
Finalizadas las pruebas de la aplicación de cálculo iterativo	28/12/15	28/12/15
Análisis de resultados	29/12/15	10/01/16
Creación de la memoria de trabajo	11/01/16	21/01/16
Entrega del trabajo	22/01/16	22/01/16

0.6.2 – Diagrama de Gantt



Nombre	Fecha de inicio	Fecha de fin
Comienzo del trabajo	19/09/15	19/09/15
Creación del plan de trabajo	19/09/15	27/09/15
Recopilación de documentación	28/09/15	4/10/15
Instalación y puesta en marcha del entorno de trabajo	5/10/15	9/10/15
Descarga de aplicaciones	5/10/15	5/10/15
Instalación y prueba de las aplicaciones	6/10/15	9/10/15
Entorno instalado y funcional	9/10/15	9/10/15
Desarrollo de la aplicación de tratamiento de logs sobre MapReduce	10/10/15	22/10/15
Consulta de la documentación	10/10/15	11/10/15
Desarrollo de la aplicación	12/10/15	20/10/15
Pruebas preliminares	21/10/15	22/10/15
Implementación de la aplicación de tratamiento de logs sobre Spark	23/10/15	6/11/15
Consulta de la documentación	23/10/15	25/10/15
Desarrollo de la aplicación	26/10/15	3/11/15
Pruebas preliminares	4/11/15	6/11/15
Pruebas de la aplicación de tratamiento de logs sobre Hadoop 1, Hadoop...	7/11/15	14/11/15
Finalizadas pruebas de la aplicación de tratamiento de logs	14/11/15	14/11/15
Desarrollo de la aplicación de cálculo iterativo sobre MapReduce	15/11/15	1/12/15
Consulta de la documentación	15/11/15	18/11/15
Desarrollo de la aplicación	19/11/15	28/11/15
Pruebas preliminares	29/11/15	1/12/15
Implementación de la aplicación de cálculo iterativo sobre Spark	2/12/15	18/12/15
Consulta de la documentación	2/12/15	5/12/15
Desarrollo de la aplicación	6/12/15	16/12/15
Pruebas preliminares	17/12/15	18/12/15
Pruebas de la aplicación de cálculo iterativo sobre Hadoop 1, Hadoop 2 ...	19/12/15	28/12/15
Finalizadas las pruebas de la aplicación de cálculo iterativo	28/12/15	28/12/15
Análisis de resultados	29/12/15	10/01/16
Creación de la memoria de trabajo	11/01/16	21/01/16
Entrega del trabajo	22/01/16	22/01/16



Nombre	Fecha de inicio	Fecha de fin
Comienzo del trabajo	19/09/15	19/09/15
Creación del plan de trabajo	19/09/15	27/09/15
Recopilación de documentación	28/09/15	4/10/15
Instalación y puesta en marcha del entorno de trabajo	5/10/15	9/10/15
Descarga de aplicaciones	5/10/15	5/10/15
Instalación y prueba de las aplicaciones	6/10/15	9/10/15
Entorno instalado y funcional	9/10/15	9/10/15
Desarrollo de la aplicación de tratamiento de logs sobre MapReduce	10/10/15	22/10/15
Consulta de la documentación	10/10/15	11/10/15
Desarrollo de la aplicación	12/10/15	20/10/15
Pruebas preliminares	21/10/15	22/10/15
Implementación de la aplicación de tratamiento de logs sobre Spark	23/10/15	6/11/15
Consulta de la documentación	23/10/15	25/10/15
Desarrollo de la aplicación	26/10/15	3/11/15
Pruebas preliminares	4/11/15	6/11/15
Pruebas de la aplicación de tratamiento de logs sobre Hadoop 1, Hadoop...	7/11/15	14/11/15
Finalizadas pruebas de la aplicación de tratamiento de logs	14/11/15	14/11/15
Desarrollo de la aplicación de cálculo iterativo sobre MapReduce	15/11/15	1/12/15
Consulta de la documentación	15/11/15	18/11/15
Desarrollo de la aplicación	19/11/15	28/11/15
Pruebas preliminares	29/11/15	1/12/15
Implementación de la aplicación de cálculo iterativo sobre Spark	2/12/15	18/12/15
Consulta de la documentación	2/12/15	5/12/15
Desarrollo de la aplicación	6/12/15	16/12/15
Pruebas preliminares	17/12/15	18/12/15
Pruebas de la aplicación de cálculo iterativo sobre Hadoop 1, Hadoop 2 ...	19/12/15	28/12/15
Finalizadas las pruebas de la aplicación de cálculo iterativo	28/12/15	28/12/15
Análisis de resultados	29/12/15	10/01/16
Creación de la memoria de trabajo	11/01/16	21/01/16
Entrega del trabajo	22/01/16	22/01/16

Capítulo 1

Evaluación del rendimiento de un computador

1.1 - Índice del capítulo 1

1.1 – Índice del capítulo 1

1.2 – Introducción

1.3 – Linpack

1.4 – Necesidad de un Nuevo modelo de evaluación

1.5 – NPB

1.6 – Conclusiones

1.7 - Bibliografía

1.2 - Introducción

Cuando hablamos de computadoras siempre nos viene a la cabeza cuál es su velocidad. Recuerdo cuando comencé en el mundo de las computadoras con mi Intel 8088, con una velocidad de reloj de 4,77 MHz (el doble en modo “turbo”), enseguida empezó una pequeña competición entre mis amigos para ver quién tenía el ordenador más rápido. Un amigo siguió con un Intel 80286 a 12,5 MHz, a lo que yo seguí, cuando pude permitírmelo, con un AMD 80386 a 40 MHz, poco después empezamos a preocuparnos por las cachés y las latencias de los chips de memoria… Por aquel entonces no teníamos muy claro de qué hablábamos, sólo que el “Secret Weapons of the Luftwafe” o el “X Wing” iban más finos cuanto mejores fuesen las características del ordenador.

Sim embargo, lo que por aquel entonces llamábamos la velocidad del ordenador va mucho más allá de un pique entre unos fanáticos de los simuladores de vuelo allá por los años 90. Desde que aparecieron las primeras computadoras su rendimiento ha sido la más importante de sus características, ya que en función del uso que se precise darle vamos a necesitar un tipo de computadora u otra.

La necesidad de una evaluación del rendimiento de las computadoras pasa tanto por el usuario final, que necesita una referencia para poder comparar con sus necesidades y decantarse por un modelo u otro, como por el fabricante, que necesita saber cómo ofrecer su producto y en qué lugar se encuentra respecto de la competencia.

Dentro de las medidas de rendimiento, tenemos unas más características:

- El tiempo de respuesta (elapsed time en inglés), que es el tiempo que tarda una aplicación en ejecutarse y que es la sensación que percibe el usuario. En un entorno multiusuario, el administrador del mismo estará preocupado por la productividad del mismo (throughput en inglés). También es importante el tiempo de CPU, que contará tanto el tiempo empleado por las instrucciones de nuestra aplicación como el empleado por el sistema operativo al ejecutarse nuestra aplicación. Variará enormemente si nuestra aplicación hace un uso intensivo de la CPU o del sistema de archivos.
- Los MIPS, o millones de instrucciones ejecutadas por segundo. Es una medida que depende de cada programa y de cada máquina. Al contrario que el tiempo, a lo largo de estas pruebas no tendremos en cuenta esta métrica. De hecho, hubiera que haber dedicado un apartado a la consecución de la misma y no creo que sea representativa para este trabajo.
- Los MFLOPS, o millones de instrucciones en coma flotante por segundo. Es una figura independiente de la máquina, pero sólo tiene sentido para aplicaciones que realizan este tipo de cálculos. Además, no todas las operaciones en coma flotante tardan lo mismo en ejecutarse, por lo que se suele emplear la figura de MFLOPS normalizados.

Además, en un sistema distribuido, es muy importante el rendimiento de las comunicaciones entre los nodos. Tan importante como la velocidad del procesador es el ancho de banda de la comunicación entre nodos, el ancho de banda de la memoria, la localidad de los datos para que estén disponibles en el nodo con los que se va a trabajar, el solicitar al sistema que se empleen menos procesadores en el nodo de los disponibles para no saturar el ancho de banda de la memoria si la aplicación hace un uso intensivo

de la misma,… Veremos a continuación cómo se realiza la evaluación de rendimiento en los súper ordenadores actuales, que no dejan de ser enormes sistemas distribuidos.

1.3 - Linpack

Si queremos saber cuáles son las computadoras más potentes del mundo, podemos consultar la página del TOP500, en http://www.top500.org.

Ordenador Tianhe-2. El más potente según el TOP500

Para realizar las pruebas de rendimiento, el TOP500 emplea **Linpack**. Linpack es un paquete de software comprendido por un conjunto de subrutinas de Fortran para resolver varios sistemas de ecuaciones lineales. Se desarrolló en el Argone National Laboratory por Jack Dongarra en 1976 [1] y está basado en otro paquete matemático, denominado BLAS (Basic Linear Algebra Subroutines).

Vemos que estamos hablando de hace 40 años, y las computadoras han cambiado bastante. De hecho, la complejidad del problema de resolución de un sistema lineal de n ecuaciones es $O(n^2)$. En principio, un sistema de 100 ecuaciones se consideraba suficientemente grande – complejidad $O(100^2)$ - . Hoy por hoy, nuestro teléfono móvil puede con problemas mucho más interesantes, por lo que el tamaño del sistema ha aumentado.

No entraré en detalle de los diferentes programas que componen el paquete, pero sí diré que hacen un uso intensivo de las operaciones en coma flotante, correspondiendo el mayor tiempo de ejecución a la rutina DAXPY de BLAS, que se limita a efectuar la operación:

y(i) := y(i) + a * x(i)

El trabajo se realiza, como es de esperar, empleando matrices que son fácilmente paralelizables, que es el motivo por el que se emplea esta rutina para medir el rendimiento de súper ordenadores.

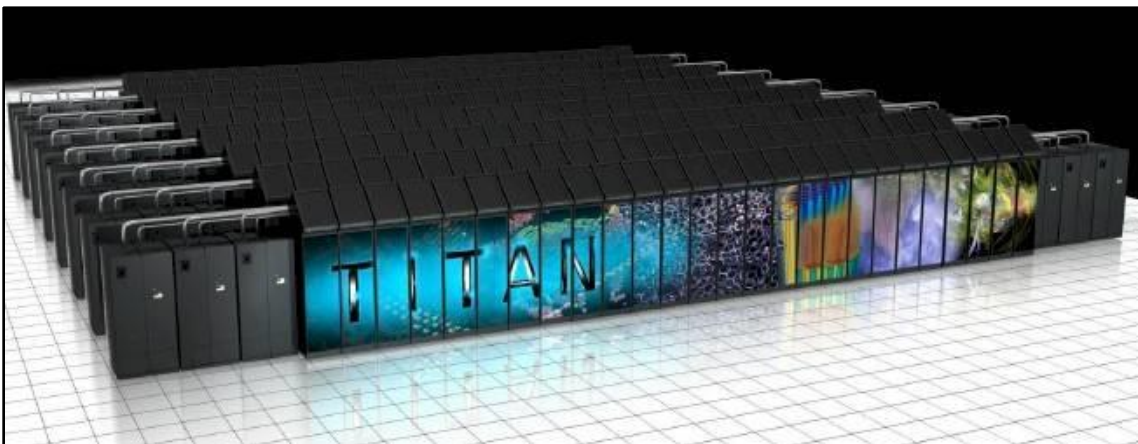
Como decimos, Linpack se desarrolló en 1976 y se ha visto ampliamente superado por otro paquete denominado Lapack (Linear Algebra Package), aunque sigue realizando operaciones intensivas en coma flotante. A pesar de ello, el TOP500 sigue empleando Linpack (una versión del mismo denominada HPL, High Performance Linpack).

Se ha dudado de la validez de Linpack como una herramienta para medir el rendimiento de estas computadoras. Por un lado, los fabricantes, al ver que era el test usado para medir el rendimiento de forma estándar, optimizaron sus máquinas para que obtuvieran una mejor clasificación en el TOP500 en lugar de optimizarlas para un rendimiento con aplicaciones reales. Por otro lado, el crecimiento en tamaño de las súper computadoras ha hecho que otros factores, además de la capacidad para ejecutar aplicaciones que hacen un uso intensivo de las operaciones en coma flotante, hayan tomado una gran relevancia en el rendimiento "real" de las aplicaciones.

1.4 – Necesidad de un nuevo sistema de evaluación

El mismo Jack Dongarra, en un informe realizado en colaboración con Michael A. Heroux para Sandia National Laboratories en 2013 plantea la necesidad de modificar el modo en que se evalúa el rendimiento de los súper computadores. En este informe [2], Dongarra y Heroux definen un conjunto de pruebas denominado HPGC (High Performance Conjugate Gradient), y que están compuestas tanto como aplicaciones que realizan un uso intensivo de la CPU como por pruebas de acceso a datos. Con ello pretenden el que la medición de rendimiento y el rendimiento percibido al ejecutar aplicaciones reales estén más en consonancia, con el objetivo de que el nuevo desarrollo por parte de los fabricantes vaya en la dirección de las necesidades reales que en la obtención de un mejor puesto en el TOP500.

En este documento los autores explican que, si bien muchas aplicaciones del mundo real emplean el tipo de cálculos que se efectúan en el Linpack, también efectúan operaciones con una menor importancia de los cálculos respecto de los accesos a datos, acceden a la memoria de forma irregular y ejecutan operaciones iterativas finas. Por lo tanto, el rendimiento medido por Linpack ofrece una imagen sesgada de la realidad.



Sistema Titan del Oak Ridge National Laboratory

Para muestra un botón, que dice la sabiduría popular. El ordenador que aparecía en el número 1 del TOP500 en Noviembre de 2012 era el sistema Titan del Oak Ridge National Laboratory. Dicho sistema posee 18.688 nodos, cada uno de los cuales tiene

un procesador AMD Opteron de 16 núcleos y una GPU Nvidia K-20. Se demostró que durante la ejecución de HPL, la totalidad las operaciones en coma flotante se realizaron en las GPUs, mientras que las aplicaciones reales que se migraron al Titan, trabajaban principalmente con las CPUs, enviando cálculos auxiliares a las K-20.

Aunque el desarrollo en computación de alto rendimiento ha hecho un gran esfuerzo para que en las aplicaciones predominen las operaciones en coma flotante de forma intensiva, siempre existirán las operaciones de los otros tipos mencionados. Los autores del informe mencionado creen que las evaluaciones del rendimiento deben reflejar esta realidad.

No quiero entrar en detalles sobre el test, sin embargo se justifica el uso del HPCG al tener en cuenta las comunicaciones necesarias al ejecutar las aplicaciones y remarcar la necesidad de emplear la localidad de datos y de mejorar el rendimiento de la memoria local para mejorar el rendimiento de las actuales computadoras.

1.5 - NPB

Como ejemplo de un modelo de evaluación de rendimiento diferente tenemos el NPB, o Nas Parallel Benchmark. Tenemos una explicación muy clara del mismoen el documento de David Bailey [3].

Fueron desarrollados a petición de la NASA en el Ames Research Center en 1991 al detectar una necesidad de tener una evaluación de rendimiento para comparar computadoras del máximo nivel. Aunque ya no se utilizan, son un claro ejemplo de qué debe analizarse para obtener una medición correcta del rendimiento de una súper computadora.

El contexto que originó esta necesidad fue la aparición de los sistemas altamente paralelos en una disciplina dominada por las computadoras vectoriales. Estos nuevos sistemas recibieron una gran publicidad, pero se necesitaba saber si representaban un gran avance sobre los sistemas vectoriales existentes. El único sistema de medición de rendimiento era Linpack que, como ya hemos dicho, se limitaba a medir la capacidad del sistema para realizar operaciones en coma flotante, pero que la NASA no consideraba representativo para las operaciones que efectuaban sus computadoras por aquel entonces.

El paquete consiste de ocho programas, especificados en un papel técnico y que daban cierta libertad a los encargados de las implementaciones. Originalmente, las ocho pruebas eran:

- EP. Embarrassingly Paralell. No precisa de comunicación entre procesos y mide el rendimiento del procesador en cada nodo.
- MG: Multi Grid. Mide la eficiencia de las comunicaciones tanto próximas como lejanas.
- CG: Conjugate Gradient. Este tipo de cálculos es típico en “grids” que precisan de comunicaciones a larga distancia irregulares.
- FT. Evalúa el rendimiento de un uso intensivo de las comunicaciones a larga distancia.

- IS. Integer Sort. Mide tanto la velocidad de cálculo como el rendimiento de la comunicación.
- LU, SP y BT. Representan la mayor parte de los cálculos que realizan un uso intensivo de la CPU.

Vemos que estos tests no sólo miden la capacidad de realizar un uso intensivo de la CPU, sino que le dan una gran importancia a las comunicaciones entre los procesos. Vemos que existe cierto paralelismo entre estos tests y lo comentado por Dongarra-Heroux en su informe. Intuitivamente también podemos ver la necesidad de medir el impacto de las comunicaciones entre procesos en un sistema de gran tamaño, ya que tiene una gran influencia en el resultado final.

1.6 - Conclusiones

Podemos ver fácilmente la necesidad de evaluar el rendimiento de una computadora, tanto a nivel de usuario final como de fabricante. El fabricante necesita por un lado saber que sus nuevos desarrollos son, efectivamente, más eficientes que los anteriores o qué cumplen las expectativas puestas en ellos, mientras que el usuario final precisa de una métrica con la que poder seleccionar el equipo que cumpla con sus necesidades.

Vemos también que en el caso de grandes sistemas distribuidos entran en juego otros factores que no son exclusivamente el rendimiento de la(s) CPU(s), sino que se necesita evaluar el rendimiento de las comunicaciones entre los procesos. Dos nodos separados por varios "switches" pueden tener un retardo realmente importante debido a la comunicación y si nuestra aplicación precisa que ambos nodos se comuniquen, este retardo puede determinar tanto o más que la potencia de las CPUs involucradas el rendimiento de la aplicación.

Es por ello que se están proponiendo nuevos sistemas de medición del rendimiento de los computadores, así como nuevas vías de investigación y desarrollo que nos ayuden a mejorar el rendimiento de dichos equipos, como la mejora de las memorias locales.

Podemos ver que esta propuesta tampoco es algo nuevo, ya que la NASA en los 90 encargó la creación de un nuevo paquete de pruebas que se adaptase mejor a las aplicaciones que empleaba ya que Linpack no proporcionaba una idea correcta del rendimiento de dichas aplicaciones sobre sus equipos.

Aunque no es el objeto de este trabajo realizar un estudio riguroso del rendimiento de un sistema concreto, sí que resulta motivador a la hora de plantearlo. No voy a poder realizar mediciones de la influencia que van a tener las comunicaciones en el desarrollo de mis pruebas, ni tengo forma de influir en ello. El sistema Hadoop es el que se encarga de todo ello, creando una capa de abstracción entre el sistema físico y yo sobre la que no tengo influencia. Sí que tengo a mi disposición tiempos totales de ejecución y tiempos de CPU que me pueden dar una idea de cómo de intensiva en el uso de la CPU es mi aplicación o si la mayor parte del tiempo se pierde en accesos a datos. De hecho, quiero ver la ineficiencia de programar una tarea iterativa en MapReduce debido al uso intensivo que se hace del sistema de archivos. En el Mare Nostrum, con la aplicación SPARK4MN puedo solicitar diferentes números de nodos para el sistema, de modo que puedo ver si se gana o se pierde eficiencia para mis aplicaciones concretas al necesitar un mayor número de comunicaciones entre nodos. Además, esta

aplicación tiene una variable de configuración que me permite solicitar que el número de switches empleados sea el mínimo posible, aunque no estudiaré esta vía por falta de tiempo.

1.7 - Bibliografía

- [1] http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf
- [2] http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf
- [3] http://www.davidhbailey.com/dhbpapers/npb-encycpc.pdf
- Módulo "Rendimiento de los computadores" de la asignatura Arquitectura de Computadoras de la UOC.

Capítulo 2

Construcción de la aplicación de tratamiento de logs para MapReduce

2.1 - Índice del capítulo 2

2.1 – Índice del capítulo 2

2.2 – Introducción: ¿Qué es Hadoop?

2.3 – Hadoop 1: MapReduce

2.3.1 – Estructura de un programa para MapReduce

2.3.1.1 – La clase Mapper

2.3.1.2 – La clase Reducer

2.3.1.3 – La clase Driver

2.4 – Hadoop 2: YARN & MapReduce 2

2.4.1 – Arquitectura de una ejecución sobre YARN

2.4.2 – YARN versus MapReduce 1

2.5 – Selección de archivos para la aplicación y funcionalidad de la misma

2.6 – Construcción de la aplicación. Comentarios al código

2.7 – Pruebas en modo local y pseudo distribuido

2.8 – Conclusiones

2.9 - Bibliografía

2.2 - Introducción: ¿Qué es Hadoop?

Formalmente, podríamos definir Hadoop como un sistema de almacenamiento y computación distribuidos. En un momento en el que las necesidades de almacenamiento y procesamiento de la información han crecido, y siguen creciendo, espectacularmente. Hadoop pretende dar solución a estas necesidades.

Hadoop es un sistema distribuido, compuesto de sistema de archivos distribuido (denominado HDFS), ofreciendo la posibilidad de paralelizar y ejecutar programas en un clúster de ordenadores.

El concepto que ofrece Hadoop es muy simple, a la par que tremendamente eficiente. La aproximación tradicional en la computación de alto rendimiento era tener código instalado en el clúster y trasladar los datos allí donde eran necesarios. Este procedimiento tiene una limitación muy importante, que es el ancho de banda de transmisión de datos entre los diferentes nodos y que supone un cuello de botella en el rendimiento de la computación distribuida. De hecho, uno de los aspectos más importantes a la hora de programar una aplicación para su ejecución en paralelo en un clúster, es el procurar que los datos estén lo más próximos que sea posible al punto donde van a ser utilizados. Hadoop le da la vuelta a esta aproximación y distribuye los datos en HDFS y mueve el código allí donde es necesario. Las aplicaciones típicas de Hadoop no tiene porqué ser especialmente voluminosas y moverlas a lo largo del clúster precisa de un ancho de banda muy inferior al necesario para trasladar los datos. Hadoop libera al programador de la tediosa tarea de distribuir los datos adecuadamente, haciéndolo por él de forma transparente.

Dentro de Hadoop podemos diferenciar varias partes componentes, entre las que distinguiremos HDFS, MapReduce y YARN, que irán apareciendo a lo largo de este trabajo. HDFS es una arquitectura maestro-esclavo distribuida, donde la función de maestro la desempeña el NameNode, mientras que la función de esclavo la desempeñan los DataNodes, siendo en estos últimos donde se almacenan los datos.

El NameNode guarda la ubicación de todos los bloques de datos del sistema de archivos, es decir, sabe en qué DataNode se encuentra cada porción de datos. Cuando un cliente, es decir nuestra aplicación o un shell, precisa trabajar con unos datos determinados es el NameNode con quien se comunica primero para conocer dónde se encuentran dichos datos. Este modelo es eficiente, y escala hasta las decenas de miles de nodos. El problema es que si el NameNode falla, perdemos todo el sistema. En Hadoop 1 este punto de fallo se solucionaba haciendo que el NameNode escribiese, de forma síncrona, sus datos en los discos locales y en un sistema de almacenamiento remoto. En Hadoop 2 diversos desarrolladores han elaborado soluciones denominadas alta disponibilidad del NameNode, que dan al sistema la capacidad de recuperarse ante un fallo catastrófico del mismo.

Una característica muy importante del sistema es que el NameNode guarda toda la información relativa a los DataNodes en la memoria RAM, por lo que este nodo del sistema debe tener una configuración de hardware diferente a la de los DataNodes, con una gran cantidad de RAM.

2.3 Hadoop 1: MapReduce

Ya he comentado que, con los volúmenes de datos que manejamos hoy día, el ancho de banda necesario para mover el código es muy inferior al ancho de banda necesario para mover los datos. Mientras que la capacidad de las unidades de almacenamiento, así como la velocidad de lectura que soportan, han estado creciendo espectacularmente, el ancho de banda de transmisión de los datos no lo ha hecho en igual medida, con una diferencia de varios órdenes de magnitud.

Es en este contexto donde aparece Hadoop y, por ende, MapReduce. MapReduce realiza un procesamiento por lotes de las diferentes peticiones que se le envían, abstrayendo la localización de los datos dentro del sistema de archivos y logrando realizar las operaciones solicitadas sobre el conjunto completo de los datos en un tiempo razonable.

El hecho de que MapReduce sea un sistema de procesamiento por lotes le imposibilita para tratar los datos de forma interactiva o procesar streams de datos que se generen en tiempo real. MapReduce es un sistema que debe utilizarse offline, sin un agente interactuando activamente con el sistema.

Para poder trabajar con MapReduce, los datos que queramos procesar deben de estar en un formato muy definido, en concreto, los datos deben de estar almacenados en pares (clave, valor).
Aunque en un principio podríamos pensar que precisamos de un tratamiento intensivo de los datos antes de poder usar MapReduce para su procesamiento, veremos que no tiene por qué ser así y que, en nuestro caso concreto de procesamiento de archivos de registro, no precisamos ningún tratamiento previo de los datos. No es en vano que se recomienda empezar a trabajar con Hadoop con aplicaciones de este tipo antes de lanzarnos a explorar en profundidad el sistema.

Lo primero que tengo que decir es que el nombre de MapReduce se debe a que el tratamiento de los datos se divide en dos partes principales diferenciadas, que son Map y Reduce. Tenemos otras etapas intermedias, como Shuffle y Order, que ya iremos viendo a lo largo de esta explicación. En ambos casos, la entrada son pares (clave, valor) y la salida son también pares (clave, valor). La clave debe ser única para cada uno de los pares.

Para el caso concreto en el que la entrada es un archivo de texto, se van tomando como entradas al mapper las diferentes líneas del archivo, siendo la clave el desplazamiento en bytes de cada una de las líneas. Es decir, si no modificamos la configuración de nuestra aplicación, al dar como entrada un archivo de texto, se considera que ya está en la forma de pares (clave, valor), siendo las claves los desplazamientos en bytes de las líneas del archivo de texto, y sus correspondientes valores las líneas en sí.

Una vez explicado el concepto de pares (clave, valor) – en adelante (K, V), podemos ver el funcionamiento de MapReduce como una serie de transformaciones sobre estos pares. En concreto, la secuencia de transformaciones la podríamos representar esquemáticamente de la forma siguiente:

(K1, V1) → (K2, List<V2>) → (K3, V3)

El conjunto de pares (K1, V1) se corresponde con la entrada al Mapper. Este realiza una transformación sobre los mismos, generando el par intermedio (K2, List<V2>). Para cada entrada, los mappers pueden generar cero, una o varias salidas. La salida de los mappers son pares clave y una lista de valores asociadas con cada una de las claves.
Cada una de las claves, con su lista de valores correspondiente, se envía a un único reducer, que se encarga de generar como salida un único par (K3, V3).

Una vez que se han generado los pares (K2, List<V2>), se realizan dos operaciones antes de pasar las entradas a los reducers correspondientes. Por un lado tenemos la operación Shuffle, que se encarga de agrupar los diferentes valores correspondientes a una misma clave a partir de los pares generados por todos los mappers. Por otro lado, se realiza la operación Sort, que ordena la lista de valores de cada una de las claves, de modo que cada reducer recibe una lista ordenada de valores K2.

2.3.1 – Estructura de un programa para MapReduce

Veremos aquí una breve descripción de la estructura de un programa de MapReduce. Cuando pasemos a comentar el código de la aplicación desarrollada citaremos de forma más detallada algunas partes del mismo que tienen un cierto interés.

A continuación describiremos muy brevemente las clases Mapper, Reducer y Driver, que son las mínimas que debe tener una aplicación para MapReduce.

2.3.1.1 – La clase Mapper

La clase Mapper es la que recibe los como entrada nuestros datos y la que va a realizar la primera de las transformaciones mencionadas anteriormente. Es obligatorio que esta clase implemente el método map(). Dicho método se llama para cada uno de los pares de entrada (K1, V1), y en él deberemos definir la transformación que se debe llevar a cabo sobre ellos, de modo que obtengamos como salida los pares (K2, List<V2>) de entrada al reducer.

2.3.1.2 – La clase Reducer

Su funcionamiento es muy similar al de la clase Mapper, y tan sólo requiere implementar el método reduce(), que es el que define la transformación que queremos realizar sobre los datos de entrada al reducer. Como en el caso anterior, se llama una vez para cada par de entrada (K2, List<V2>).

2.3.1.3 – La clase Driver

Aunque las clases Mapper y Reducer son las que efectúan las tareas que nuestra aplicación debe llevar a cabo sobre los datos, es necesaria una clase más para comunicarse con el entorno de Hadoop. Este trabajo lo lleva a cabo la clase Driver.

La clase Driver es la que se encarga de decirle a Hadoop cuales van a ser las clases Mapper y Reducer, dónde están los datos de entrada y dónde almacenar los datos de salida, así como el formato que debe de darse a los mismos.
Es también aquí donde estará el método main() de nuestra aplicación Java.

2.4 - Hadoop 2: YARN & MapReduce 2

Ya hemos comentado que MapReduce es un sistema de procesamiento por lotes que no está preparado para que un agente interactúe activamente con el sistema. Las necesidades que han ido apareciendo precisan de una interacción en tiempo real con los datos que va generando el sistema, o con los datos enviados por agentes en otros sistemas.
Es por ello que Hadoop, aunque perfectamente válido para muchas aplicaciones, necesitaba adaptarse de modo que fuese capaz de procesar streams de datos generadas en tiempo real.

En principio, el objetivo de YARN, que es el acrónimo de Yet Another Resource Manager, era mejorar la implementación de MapReduce, pero es de un carácter lo suficientemente general como para que pueda adaptarse a otros paradigmas de computación distribuida, como el procesado de streams mencionado.

Aunque en el apartado anterior no he querido entrar en la arquitectura de Hadoop, sí me parece interesante el realizar una comparación de las arquitecturas de Hadoop 1 y Hadoop 2. En la figura 2.1[1] podemos ver una comparación de ambas arquitecturas.

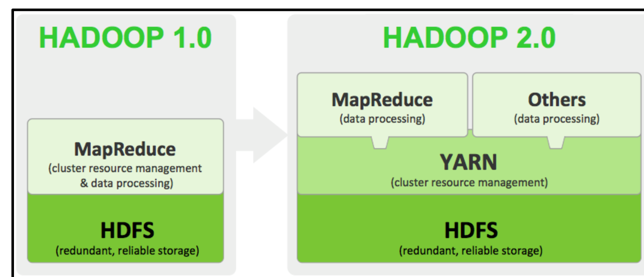


figura 2.1 – arquitecturas de Hadoop 1 y 2

En la figura podemos ver como en Hadoop 1 MapReduce, además de encargarse del proceso de datos, también se encargaba de la gestión de los recursos del clúster. En Hadoop 2, en aras de la eficiencia y la versatilidad, se separan estos dos conceptos. Seguimos teniendo como base al sistema de archivos distribuido HDFS, pero sobre él ahora tenemos a YARN, como un gestor de recursos genérico sobre el que pueden ejecutarse una variedad de modelos de programación, entre los que se encuentran MapReduce y Spark, como veremos más adelante.

El modelo de programación de MapReduce 2 no varía significativamente respecto a su predecesor, siguiendo las aplicaciones la misma estructura. De hecho, la aplicación de tratamiento de archivos de registro objeto de este capítulo es la misma en tanto para Hadoop 1.2.1 como para Hadoop 2.7.1.
No obstante, hay variaciones entre las versiones 1 y 2, y las versiones de Hadoop 2 nos permiten todo tipo de combinaciones, como trabajar con aplicaciones escritas para MapReduce 1 sobre YARN y viceversa. En este trabajo, sin embargo, no jugaré con esta posibilidad y mantendré las configuraciones por defecto.

[1] La figura se ha obtenido de la web http://www.tomsitpro.com/articles/hadoop-2-vs-1,2-718.html

2.4.1 – Arquitectura de una ejecución sobre YARN

También me parece interesante en este caso el comentar brevemente la arquitectura de una ejecución sobre YARN, para que veamos qué sucede entre bambalinas mientras esperamos los resultados de la aplicación que hemos enviado al sistema.
Soy consciente de que estoy detallando más YARN que MapReduce 1. Por un lado, como ya he comentado, lo explicado para MapReduce 1 es válido para su versión 2, mientras que al ser YARN la arquitectura que se emplea actualmente, me parece oportuno el entrar más en su detalle.
La figura 2.2[2] nos muestra esta arquitectura.

En un clúster sobre el que tengamos instalada una versión de Hadoop vamos a encontrarnos con un único gestor de recursos, llamado ResourceManager. El cliente del sistema se comunica con el ResourceManager y le da la aplicación a ejecutar.

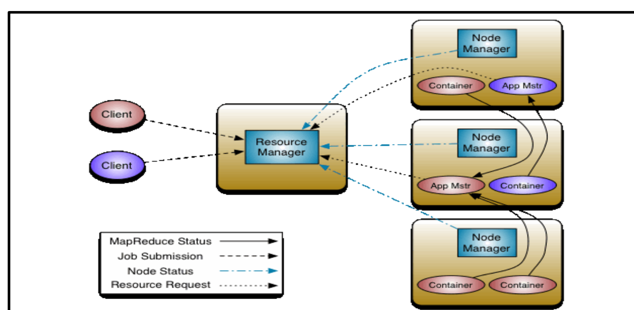


figura 2.2 – arquitectura de ejecución de MapReduce

En cada uno de los nodos del clúster tenemos un NodeManager, que es el segundo de los demonios de larga vida que existen en un clúster que implemente YARN, y que se encarga de gestionar los recursos del nodo. Una vez que el cliente ha pasado la aplicación al ResourceManager, éste busca un nodo, comunicándose con los NodeManagers del sistema, que pueda ejecutar el master de la aplicación en un contenedor. Una vez que el master de la aplicación se está ejecutando, se puede llegar al final de la ejecución dentro de este único contenedor, o pueden ser necesarios más. En este último caso, el NodeManager se comunica con el ResourceManager para pedirle más recursos. Una vez que éste se los ha concedido, el NodeManager se comunica con los NodeManagers de los nodos que le ha concedido el ResourceManager.

Podemos ver también en la figura, que los NodeManagers se comunican con el ResourceManager para que éste tenga conocimiento del estado de todos ellos y pueda gestionar sus recursos.

Para terminar con este apartado mencionaré que, puesto que no existe una política de planificación idónea para todos los tipos de trabajo, YARN nos permite elegir entre tres, de modo que podamos adaptar la planificación a nuestro trabajo. Las tres políticas son FIFO, Capacity Scheduler y Fair Scheduler, aunque no las trataré en detalle.

---

[2] Imagen tomada de la web https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

2.4.2 – YARN versus MapReduce 1

En MapReduce 1 tenemos dos demonios que controlan la ejecución de los trabajos. Por un lado tenemos un jobtracker y uno o más tasktrackers. El jobtracker se encarga de coordinar el trabajo de los diferentes tasktrackers, además de repartir los recursos del sistema entre ellos. Además, el jobtracker se encarga de monitorizar el desarrollo del trabajo en los diferentes tasktrackers, reiniciándolos si alguno falla o su trabajo se está desarrollando demasiado lentamente.

En YARN estas labores se llevan a cabo por entidades diferentes, como son el gestor de recursos y master de la aplicación, y su diseño está orientado a resolver las limitaciones presentadas por MapReduce en su implementación original.

2.5 - Selección de archivos para la aplicación y funcionalidad de la misma

Para poder realizar las pruebas necesarias que nos permitan cumplir con los objetivos de este trabajo, precisaba de archivos que cumpliesen una serie de requisitos. Puesto que las aplicaciones que voy a generar en primer lugar son aplicaciones de tratamiento de archivos de registro, necesitaba archivos de log de acceso público.
Además, dichos archivos debían de estar estructurados de modo que fuese posible el tratarlos como pares (clave, valor) sin una necesidad grande de tratamiento de los mismos, para perder el menor tiempo posible en dicho tratamiento, y maximizar el tiempo dedicado al estudio objeto de este trabajo. La mayoría de los archivos de registro de los servidores cumplen con estas características, puesto que son líneas formadas por campos de datos, separados por comas, espacios o guiones.

Son muchos los archivos de datos de acceso público que podemos obtener en la red. Una de las limitaciones con las que me he encontrado son los diversos formatos de los mismos, ya que me interesaba que dichos archivos estuviesen en texto plano para poder importarlos directamente a un tipo Text. Tras una búsqueda bastante intensiva, me decidí por unos archivos antiguos de la NASA disponibles en la dirección:

http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

Son archivos correspondientes a los registros de los accesos a los servidores de la NASA en los meses de Julio y Agosto del año 1995. En ellos tenemos diversos campos, aunque los que me interesan son el primero, que es la URL desde la que se ha accedido al servidor, y el último, que son los bytes descargados desde la página. Entre ellos tenemos varios como son el tipo de petición realizada al servidor (GET en todos ellos), y el código con el que se ha respondido al cliente (200 OK en la mayoría de los casos). Dichos campos están separados por espacios.

La funcionalidad de la aplicación de tratamiento de logs va a ser encontrar el mayor número de bytes descargados por cada uno de los diferentes clientes, ordenados por IP (o URL, dependiendo de cómo esté registrado).

Aunque existe un gran número de clientes diferentes, la mayoría se repite un gran número de veces y ha descargado un número diferente de bytes en cada acceso. Esta es una aplicación bastante típica de Hadoop y requiere un solo pase Map → Reduce. Es habitual que las aplicaciones más complejas no consigan sus objetivos en un solo paso, sino que tengan que realizar varios, siendo la salida del reducer de un ciclo la entrada al mapper del siguiente.

Otro problema con el que me enfrentaba a la hora de seleccionar los archivos era su tamaño. No sólo precisaba de un tamaño grande para que el sistema tardase un tiempo apreciable que me permitiese detectar diferencias en los tiempos de ejecución, sino que Hadoop, y HDFS en concreto, están optimizados para un tamaño de archivos grande (a partir del orden de los GB).
Los dos archivos seleccionados suman un total de algo menos 3 millones y medio de líneas. Aunque pueda parecer mucho, el tiempo de procesado de los mismos se mide en segundos y no me parecía suficiente. Es por ello que los he unido en un mismo archivo y he copiado dos veces los datos, quedando un total de unas 13.800.000 líneas y un tamaño de archivo de 1,5 GB.

Aunque el tiempo de procesado sigue sin ser muy elevado (en torno al medio minuto en Hadoop), creo que puede ser suficiente para obtener resultados significativos. En caso de que las pruebas revelen lo contrario, siempre se pueden hacer copias del archivo con nombres diferentes, de forma que tengamos un cierto número de archivos iguales de tamaño elevado que nos provean de una salida suficientemente significativa.

Debemos tener presente en mente en todo momento el objeto de este trabajo, que no es otro que comparar el rendimiento de las diferentes implementaciones de Hadoop y Spark. Es por ello que podemos trabajar sobre archivos iguales para tener un tamaño en la entrada suficientemente grande como para que el estudio sea significativo, ya que lo que nos interesa es, precisamente, el tamaño en la entrada y no su contenido.

La salida que vamos a obtener es la misma tanto con los archivos originales, como con un número indeterminado de archivos incrementados de la forma explicada, lo cual podría servirnos como comprobación de la corrección del proceso. De todos modos, los tres millones y medio de accesos a la web de la NASA se han realizado por unos 137.000 clientes diferentes, lo cual hace que la comprobación de la igualdad de los resultados deba de realizarse de forma automatizada (¿con una aplicación para Hadoop?) si no queremos que nos lleve un tiempo excesivo.

Puede verse que ya he realizado una serie de pruebas preliminares que me permiten tener cierto conocimiento de las aplicaciones y de los resultados, pero no adelantemos acontecimientos.

2.6 - Construcción de la aplicación. Comentarios al código

La aplicación está escrita en Java y, como ya he comentado en el apartado 2.3, la aplicación debe contener como mínimo tres clases; el mapper, el reducer y el driver. Aunque puede hacerse en tres archivos diferentes, lo he hecho en un único archivo, ya

que no es un código extenso y, al tenerlo todo en el mismo archivo, el pasar de una parte a otra del mismo me resulta más cómodo.

Paso ahora a comentar el código de las diferentes clases, lo cual nos permite no sólo ver la implementación del trabajo, sino también profundizar en la estructura de las aplicaciones para MapReduce.

Aparte de los elementos Java estándar necesarios, necesitaremos importar una serie de bibliotecas específicas de Hadoop para poder realizar la aplicación. En ellos se incluyen características como las implementaciones de las clases Mapper y Reducer, los tipos de datos propios de Hadoop que vamos a utilizar en la aplicación, las implementaciones para manejo de archivos del HDFS y la configuración del contexto.

No transcribiré el código de la aplicación, porque sería largo y tedioso. Tan solo comentaré algunas partes específicas del mismo. El resto puede verse en los archivos adjuntos al trabajo, y he procurado comentarlo lo máximo posible para que se pueda seguir fácilmente.

La declaración de la clase Mapper sería la siguiente:

public static class LogMapper extends Mapper<Object, Text, Text, IntWritable>

Se declara como una extensión de la clase Mapper de Hadoop. Podemos ver que está declarada con genéricos, cosa característica de Hadoop. Los dos primeros parámetros son los tipos del par (clave, valor) de entrada, mientras que los otros dos son los tipos del par (clave, valor) de salida. Se puede ver que la clave del par de entrada se declara como Object.

Además, debemos declarar el método map(), que se llama para cada par (K, V) de entrada.

public void map (Object key, Text value, Context context) throws IOException, InterruptedException

Los parámetros son los tipos del par de entrada y el contexto. Recordemos que la clase Context es la que nos permite comunicarnos con el entorno de Hadoop.

El resto de la clase es el tratamiento del par de entrada hasta convertirlo en el par de salida. Aparte de los tipos de datos específicos de MapReduce, no tenemos código diferente del Java estándar, y se puede seguir fácilmente.

La definición de la clase Reducer es la siguiente:

public static class LogReducer extends Reducer<Text, IntWritable, Text, IntWritable>

Se declara como una extensión de la clase Reducer. Al igual que con la clase LogMapper, vemos que declaramos los tipos de los pares (K, V) de entrada y salida. En este caso ambos pares son iguales.

Debemos crear el método reduce() que, como el caso de map(), se llama una vez para cada par (K, V) de entrada a la clase.

public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException

Como ya he comentado para la clase Mapper, el resto es código bastante estándar de Java y se puede seguir con facilidad.

De la clase Driver, creo interesante comentar lo siguiente:

Configuration conf = new Configuration();

Establecemos la clase Configuration. Esta clase se encarga de leer los archivos de configuración de Hadoop, los cuales podemos modificar, para crear la configuración del trabajo que se va a llevar a cabo. También podemos modificar dicha configuración en tiempo de ejecución.

Job job = Job.getInstance(conf, "hadoop_log");

Creamos el trabajo para que Hadoop lo ejecute. Le pasamos como parámetros la configuración del mismo y un nombre descriptivo.

job.setJarByClass(HadoopLog.class);

Establecemos el archivo jar empleado hallando de donde proviene la clase proporcionada.

job.setMapperClass(LogMapper.class);
job.setReducerClass(LogReducer.class);

Le decimos al entorno de Hadoop cuáles son las clases Map y Reduce de la aplicación.

job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);

Le informamos a Hadoop de los tipos de datos a los que pertenecen los elementos del par de salida.

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

Determinamos dónde leer la entrada y dónde escribir la salida a partir de los argumentos de la aplicación.

System.exit(job.waitForCompletion(true)? 0 : 1);

Esperamos a que termine el trabajo.

2.7 - Pruebas en modo local y pseudo distribuido

Ya vimos en el apartado 1.4 del capítulo anterior cuál es el comando a ejecutar en la terminal para ejecutar una aplicación escrita para MapReduce y empaquetada en formato jar. No entraré en detalle de cómo se compila la aplicación y se empaqueta en jar para no cargar el texto. Tan sólo comentaré que Hadoop trae por defecto una aplicación para poder efectuar este empaquetamiento en formato jar.

He comentado en el apartado 2.5 de este mismo capítulo que he creado un archivo de 1,5 GB de peso y casi 14 millones de líneas de texto. Para probar el correcto funcionamiento de la aplicación he creado un archivo de tan sólo 15000 líneas, que se ha mostrado más que suficiente para este propósito.

Una parte del archivo de salida es:

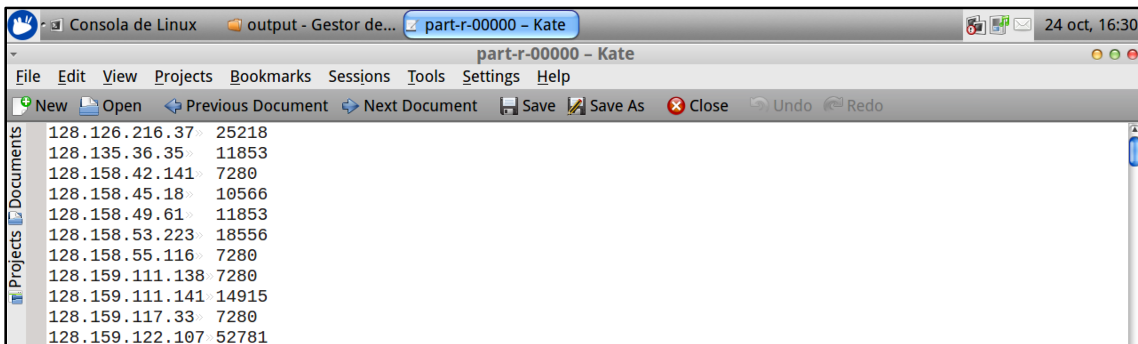


Figura 2.3 – salida de HadoopLog

En él se puede ver, de forma ordenada por IP (o nombre de la URL desde la que se ha accedido a la página de la NASA) el número máximo de bytes que se han descargado en todos los accesos hechos desde dicha IP. He comprobado minuciosamente que, efectivamente, el resultado es el deseado.

Si nos vamos al resultado presentado al final de la ejecución por la consola, obtenemos el mostrado en la figura 2.4 (se ve una pequeña parte). Vemos una serie de métricas interesantes y muy importantes en el análisis que vamos a desarrollar a lo largo de este trabajo, como los bytes leídos, los bytes escritos, el número de entradas al mapper y el número de entradas al reducer, que no es otro que el número de URLs diferentes existentes en el archivo de entrada.
Quiero llamar la atención sobre una métrica fundamental para este trabajo. Es la métrica CPU time spent (ms), que no es sino el tiempo de procesador empleado por la aplicación dado en mili segundos. Podemos ver que este tiempo es cero algo que, evidentemente, no está bien. Determinadas métricas están gestionadas por el JobTracker (uno para cada trabajo que enviamos a Hadoop), que a su vez gestiona los diferentes TaskTrackers (uno para cada tarea en las que se divide el trabajo). Estos

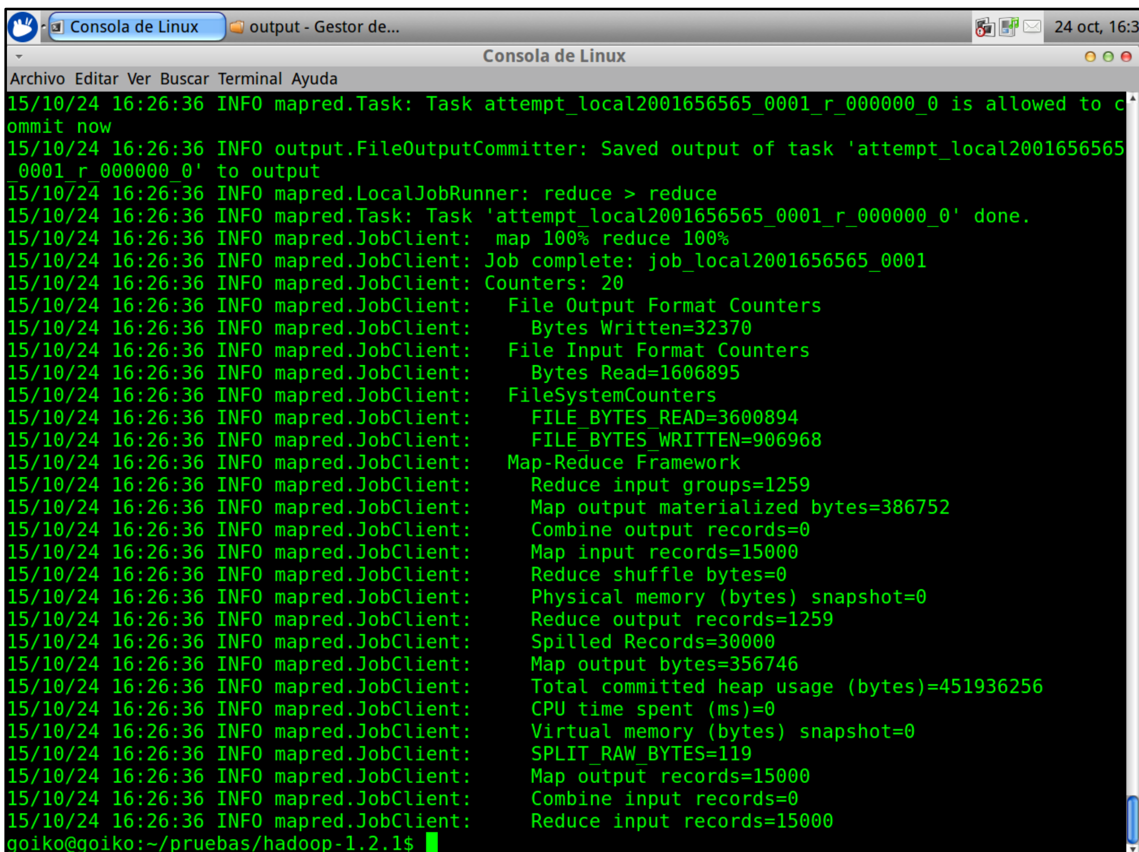


figura 2.4 – métricas de MapReduce

trackers están disponibles en modo de servidor, a los que se puede acceder desde un navegador. El problema es que dichos trackers sólo se inician en modo pseudo distribuido (o en modo distribuido cuando trabajamos en un clúster). Es por ello que necesitamos ejecutar las pruebas en modo pseudo distribuido si queremos obtener tan preciada métrica. Además, la interfaz de usuario accesible desde el navegador nos permite ver gráficas de los tiempos ejecución, además de otras y nos da la información de una forma mucho más amable.
Otro dato que echamos de menos, esta vez porque simplemente no aparece, es el tiempo total de ejecución, y que sí aparece en el modo pseudo distribuido. En su momento veremos que hay una gran diferencia entre el tiempo de CPU y el tiempo total de ejecución. Esta diferencia es la latencia del sistema, y es el precio que debemos pagar por emplear Hadoop. Sin embargo, son unos pocos segundos, mientras que Hadoop permite ejecutar en minuto trabajos que llevarían horas de otro modo.
Es por la necesidad de tener estas métricas que trabajaremos en modo pseudo distribuido, y no emplearemos más el modo local.

La configuración del modo pseudo distribuido es un proceso bastante tedioso, en el cual debemos modificar una serie de archivos de configuración de Hadoop. No lo mostraré ya que existe abundante documentación al respecto, habiendo seguido para este trabajo la oficial de la página de Apache.

Los archivos a modificar son:

- core-site.xml
- hdfs-site.xml

- mapred-site.xml
- yarn-site.xml (sólo Hadoop 2)

Dependiendo de si estamos trabajando con Hadoop 1 ó 2, estos archivos se encontrarán en diferentes subdirectorios, pero su formato es idéntico.

Tras asegurarnos de que podemos crear una conexión SSH sin necesidad de introducir claves, y de asegurarnos de la existencia de la variable de entorno JAVA_HOME, deberemos formatear el HDFS e iniciar los servicios necesarios para poder acceder a las métricas de las diferentes ejecuciones. Para ello ejecutaremos:

Hadoop 1
$ hadoop namenode –format
$ start-all.sh

Hadoop 2
$ hdfs namenode –format
$ yarn-daemon.sh start resourcemanager
$ yarn-daemon.sh start nodemanager
$ hadoop-daemon.sh start namenode
$ hadoop-daemon.sh start datanode
$ mr-jobhistory-daemon.sh start historyserver

Con todo configurado adecuadamente, pasamos a probar la ejecución de la aplicación en modo pseudo distribuido. En principio, una aplicación que funcione en modo local no debe presentar ningún problema de ejecución en modo pseudo distribuido o en modo clúster. Lo que vamos a probar realmente es que la configuración del sistema es la correcta y cómo debemos visualizar las métricas.

Los directorios de trabajo se deben crear en el HDFS. Tampoco daré detalles al respecto, por existir abundante documentación al respecto y por no ser el objetivo de este trabajo. Salvo esta diferencia, la forma de ejecutar la aplicación es igual al modo local. El resultado es, también, muy parecido al obtenido en la ejecución local con la salvedad de que la métrica time spent (ms) sí nos dará un valor, como podemos ver en la figura 2.5.

```
15/10/25 16:58:56 INFO mapred.JobClient:     Total committed heap usage (bytes)=555745280
15/10/25 16:58:56 INFO mapred.JobClient:     CPU time spent (ms)=5050
15/10/25 16:58:56 INFO mapred.JobClient:     Combine input records=0
15/10/25 16:58:56 INFO mapred.JobClient:     SPLIT_RAW_BYTES=104
15/10/25 16:58:56 INFO mapred.JobClient:     Reduce input records=15000
15/10/25 16:58:56 INFO mapred.JobClient:     Reduce input groups=1259
15/10/25 16:58:56 INFO mapred.JobClient:     Combine output records=0
15/10/25 16:58:56 INFO mapred.JobClient:     Physical memory (bytes) snapshot=541970432
15/10/25 16:58:56 INFO mapred.JobClient:     Reduce output records=1259
15/10/25 16:58:56 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=2790158336
15/10/25 16:58:56 INFO mapred.JobClient:     Map output records=15000
goiko@goiko:~/programas/hadoop-1.2.1$
```

figura 2.5 – detalle de las métricas de MapReduce en modo pseudo distribuido

Si queremos una vista de las estadísticas de la ejecución en un formato más amable, no tenemos más que ir al navegador y escribir en la barra de direcciones localhost:50030. Veremos aquí información procedente del TaskTracker y del JobTracker correspondiente a los trabajos en ejecución actualmente y los trabajos ejecutados con anterioridad. Entraré en más detalle cuando haga las pruebas de rendimiento, pero si hacemos click en el trabajo recién completado, en el apartado de

Completed Jobs tendremos a nuestra disposición una gran cantidad de información, como el tiempo total de CPU, el número de tareas dedicadas al map, el número de tareas dedicadas a reduce y el tiempo de CPU de cada una de ellas por separado, entre otras valiosísimas métricas.

A pesar de que sólo veremos los trabajos ejecutados en esta sesión, si hacemos click en Local Logs, job Tracker History tendremos acceso las métricas de todos los trabajos ejecutados. Es más, esta es una interfaz de un servicio al que podemos acceder programáticamente mediante comandos GET, lo que nos será muy útil a la hora de presentar las comparaciones entre Hadoop y Spark.

En la figura 2.6 podemos ver una vista de cómo se presentan los datos en el navegador. Como ya he dicho, ahondaré en los datos proporcionados por el servicio cuando llegue el momento. Sirva ahora tan sólo como curiosidad y para ver que las métricas se separan en totales, métricas parciales para map y métricas parciales para reduce.

	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	3.883
	Launched reduce tasks	0	0	3
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	1
	Data-local map tasks	0	0	1
	SLOTS_MILLIS_REDUCES	0	0	25.304
File Output Format Counters	Bytes Written	0	0	32.110
File Input Format Counters	Bytes Read	0	0	1.606.895
FileSystemCounters	FILE_BYTES_READ	0	386.764	386.764
	HDFS_BYTES_READ	1.606.999	0	1.606.999
	FILE_BYTES_WRITTEN	441.773	551.314	993.087
	HDFS_BYTES_WRITTEN	0	32.110	32.110

figura 2.6 – métricas de MapReduce en el UI

Esta prueba la he efectuado con Hadoop 1.2.1. El proceso para Hadoop 2.7.1 es análogo, con alguna diferencia en los comando de HDFS y en los puertos mediante los que se accede a la interfaz a través del navegador. No concretaré dichas diferencias por no sobrecargar el trabajo, pues son mínimas. De hecho, podemos emplear los mismos comandos para el HDFS, aunque se nos avisa de que están obsoletos y se nos recomienda usar los nuevos.

2.8 - Conclusiones

Puede verse que el código de la aplicación descrita no es ni extenso ni complejo. Ateniéndonos a una estructura bastante simple, podemos construir rápidamente una aplicación para MapReduce plenamente operativa, dejando toda la parte de la administración del clúster a Hadoop.

Podemos ver también que la misma aplicación la podemos ejecutar tanto en Hadoop 1 sobre el MapReduce original, como en Hadoop 2 en donde se ejecuta sobre YARN, que es una tecnología de administración de clúster totalmente diferente y mucho más polivalente.

No era el objetivo de este capítulo el ahondar en el modelo de programación de MapReduce ni complicar el código. El objetivo era realizar una toma de contacto con el modelo de programación, elaborar una aplicación con una funcionalidad determinada y realizar una serie de pruebas que nos permitiesen tanto comprender el trabajo efectuado como tomar contacto con el sistema de empaquetado en formato jar de las aplicaciones MapReduce y el modo de ejecución de las mismas.

Encuentro engañosa, no obstante, la simplicidad del código presentado, ya que el conjunto de conceptos necesario para una completa comprensión del mismo es complejo y extenso. Además, la familiarización con un entorno como es Hadoop, su modelo de programación, su modo de ejecución y el acostumbrarse a la forma de acceder a las diferentes métricas es complejo y trabajoso (más trabajoso que complejo, quizás). Bien es cierto también que sin profundizar excesivamente en dichos conceptos se puede construir una aplicación funcional, pero no es como yo entiendo ni la elaboración de un TFG ni ningún aspecto de mi vida.

También llevan su tiempo las pruebas, así como el depurado, ya que los mensajes de error que proporciona Hadoop son bastante crípticos, limitándose a mencionar la excepción que se ha producido en la ejecución. Si queremos algo más elaborado, lo tenemos que construir nosotros.

A pesar de lo dicho, el trabajo realizado ha sido una tarea positiva y satisfactoria, habiendo encontrado incluso placentero el adentrarme en este nuevo mundo de la computación distribuida.

2.9 - Bibliografía

- Turkington, Garry. Hadoop Beginner's Guide. Birmingham: Packt, 2013.
- Holmes, Alex. Hadoop in Practice. 2ª ed. Shelter Island, NY: Manning, 2015.
- Parsian, Mahmoud. Data Algorithms: Recipes for Scaling up with Hadoop and Spark. Sebastopol, CA: O'Reilly, 2015.
- White, Tom. Hadoop, The definitive Guide. 4ª ed. Sebastopol, CA: O'Reilly, 2015.
- White, Tom. Hadoop, The definitive Guide. 2ª ed. Sebastopol, CA: O'Reilly, 2011.
- Documentación oficial de Apache Hadoop desde:

  https://hadoop.apache.org/docs/stable/

Capítulo 3

Construcción de la aplicación de tratamiento de logs para Spark

3.1 - Índice del capítulo 3

3.1 – Índice del capítulo 3

3.2 – Introducción. ¿Por qué Spark?

3.3 – Modelo de programación de Spark

- **3.3.1 – RDDs.**
- **3.3.2 – Spark es “vago”**
- **3.3.3 – Arquitectura de Spark**

3.4 – Construcción de la aplicación. Comentarios al código

3.5 – Prueba de la aplicación

3.6 – Conclusiones

3.7 - Bibliografía

3.2 - Introducción. ¿Por qué Spark?

Podemos decir que Spark es un entorno de computación distribuida para el tratamiento de datos a gran escala. Dicho así, no parecen existir diferencias entre la definición que dimos de Hadoop en el capítulo anterior y la propia definición de Spark.
Una diferencia es que Spark no emplea el motor de MapReduce para ejecutarse, a pesar de tener ciertos parecidos. Es más, como ya comenté en el capítulo 1, Spark precisa de un gestor de clúster, que puede ser YARN o no. Asimismo, Spark puede integrarse para funcionar sobre HDFS, pero no tiene porqué ser así, y podemos integrar Spark en otro sistema de archivos distribuido.

Otra gran diferencia es que Spark puede mantener grandes conjuntos de datos en memoria entre trabajos. Como ya veremos en el caso del trabajo iterativo, Hadoop debe guardar en disco los datos de salida de cada iteración y que son a su vez la entrada de la siguiente. La capacidad de Spark de no tener que realizar el guardado de dichos datos en disco le permite obtener, en teoría, un rendimiento un orden de magnitud superior al de Hadoop. Uno de los objetivos de este trabajo es comprobar hasta qué punto bate Spark a Hadoop (o no) en determinados trabajos.

Además, la API de Spark es mucho más rica que la de MapReduce, a la que lastra su concepción original, lo que permite una experiencia de usuario superior, a la par que aplicaciones más potentes a la hora de trabajar con nuestros datos.

Aunque fuera del alcance de este trabajo, también quiero comentar que Spark se ha mostrado como una excelente plataforma para el desarrollo de herramientas de análisis, como aprendizaje computacional y procesamiento de gráficos, algo impensable en el modelo de MapReduce.

3.3 - Modelo de programación de Spark

3.3.1 – RDDs

RDD es el acrónimo de Resilient Distributed Dataset, y es la estructura de datos principal en Spark. Un RDD es, simplemente, una colección de objetos, inmutable y distribuida.
El que sea una colección distribuida quiere decir que cada RDD puede estar dividido en diferentes particiones, cuyos datos pueden ser empleados en diferentes nodos del clúster.

Los RDDs pueden contener objetos de los tres lenguajes de programación admitidos por Spark; Scala, Java y Python, así como clases definidas por el usuario.

Un usuario puede crear RDDs de dos formas diferentes. Por un lado, puede cargar un conjunto de datos externo en un RDD, o bien puede transformar una estructura de datos existente en un RDD.

3.3.2 – Spark es “vago”

Quizás esta afirmación no sea la mejor a la hora de promocionar el uso de Spark, sin embargo veremos que lo es por motivos de rendimiento. Antes de entrar en detalle en el porqué de la pereza de Spark, hablaremos de las dos operaciones que podemos ejecutar sobre los RDDs; transformaciones y acciones.

Las transformaciones construyen un nuevo RDD a partir de uno ya existente. Un ejemplo clásico a la hora de explicar una transformación en Spark es filtrar los datos existentes en un RDD que cumplen una condición.

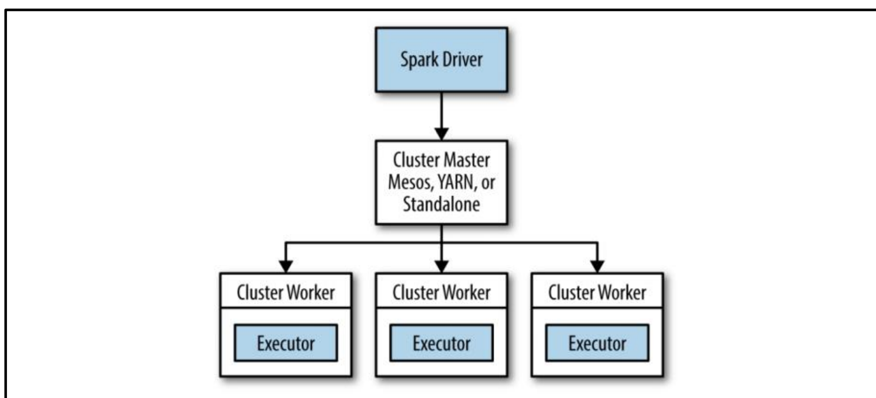
Las acciones, por otro lado, calculan un resultado a partir de los datos contenidos en un RDD y devuelven dicho resultado al Driver o lo almacenan en un sistema externo. Una acción sencilla que podemos ejecutar sobre un RDD es *first()*, que nos devuelve el primer elemento del RDD.

La diferencia entre transformaciones y acciones radica en la forma en que Spark computa los RDDs. Si bien podemos definir nuevos RDDs en cualquier momento, Spark sólo los materializará en el momento en que una acción los necesite. Es lo que se denomina *lazy execution model*, y tiene sentido en el ámbito del Big Data. Debemos tener en mente que Spark es un entorno creado para trabajar con grandes conjuntos de datos. Si materializamos los RDDs es el preciso momento de su definición, o cuando efectuamos una transformación sobre uno de ellos, enseguida comprometeremos los recursos del sistema debido al gran tamaño de los datos con los que trabajamos. Spark, sin embargo, al ser vago, una vez que le pedimos que ejecute una acción, calculará tan sólo los datos necesarios de toda la cadena de transformaciones previa para hallar el resultado final.
Este modelo de ejecución no sólo permite ahorrar en recursos de almacenamiento, sino que hace que el rendimiento sea mayor al calcular sólo aquello que se necesita.

3.3.3 – Arquitectura de Spark

Daré una explicación breve y los más sencilla posible de la arquitectura de ejecución de Spark. El objeto del trabajo no es ahondar en este aspecto, a la par que existe gran cantidad de información al respecto. Evidentemente, la explicación que daré será para el modo de ejecución en modo distribuido.

La siguiente figura está tomada de “Learning Spark. Lightning Fast Big Data Analysys”, de O’Reilly.



Arquitectura de Spark

En modo distribuido, Spark emplea una arquitectura maestro/esclavo con un gestor central y muchos nodos de trabajo distribuidos. El coordinador central se denomina Driver, y se comunica con una gran cantidad de nodos trabajadores, denominados executors.

Como sucedía en MapReduce, el Driver es el proceso donde se ejecuta el método main() de nuestra aplicación Java, y que corre en su propia JVM. El driver cumple dos cometidos:

- Convertir el programa de usuario en tareas que se puedan distribuir a lo largo del clúster.
- Planificar dichas tareas en los executors.

Aunque resulta evidente, los executors se encargas de llevar a cabo las tareas en las que el Driver ha dividido el programa de usuario.

Vemos, finalmente, que el Driver se comunica directamente con el sistema de gestión del clúster que, como ya hemos comentado, puede elegirse entre varios.

3.4 - Construcción de la aplicación. Comentarios al código

Como ya he hecho en el caso de la aplicación de gestión de logs para MapReduce, comentaré brevemente el código de la aplicación equivalente en Spark. Como puede verse en el archivo fuente de Java que se adjunta, el código es mucho más compacto y potente que el de MapReduce.
Puesto que Java 7 no implementa las clases anónimas ni las expresiones lambda, debemos incluir las implementaciones de las clases Function y Function2 de Spark.

Lo primero que hacemos es leer el archivo donde se encuentran los datos y pasarlo a un RDD. Debemos recordar que los RDDs son las estructuras de datos distribuidas básicas para trabajae en Spark. La línea que lo hace es:

```
JavaRDD<String> lines = sc.textFile("/home/goiko/datos/log.txt");
```

Posteriormente, a partir de este RDD y mediante la transformación map(), creamos un nuevo RDD del tipo JavaPairRDD. Este tipo de RDD imita el formato (K, V) de MapReduce y nos permite trabajar de forma análoga a como lo haríamos con este modelo de programación. Pienso que la parte del código correspondiente a la creación del nuevo RDD es intersante y la transcribo a continuación:

```
JavaPairRDD<String, Integer> reducerInput = lines.mapToPair(new
PairFunction<String, String, Integer>()
                {

                public Tuple2<String, Integer> call (String s) {
                …
```

Debemos implementar la interfaz PairFunction, así como la clase Tuple2 de Spark. No entraré en detalles este código. Tan sólo comentar que pasamos del RDD estándar lines, a un nuevo RDD en formato (K, V) denominado reducerInput.

Dentro de esta implementación podemos ver código Java estándar muy similar al de la aplicación para MapReduce, ya que la transformación que debemos hacer sobre los datos de entrada es la misma.

Una muestra de la potencia del código de Spark es la implementación del Reducer, que transcribo a continuación:

```
JavaPairRDD<String, Integer> reduceResult = reducerInput.reduceByKey(

                new Function2<Integer, Integer, Integer>()
                {
                        public Integer call(Integer i1, Integer i2)
                        {
                                if (i1 > i2)
                                        return i1;
                                else
                                        return i2;
                        } });
```

Tan sólo debemos aplicar la transformación reduceByKey a nuestro PairRDD, con una función de reducción que seleccione el valor más alto de la lista de valores que corresponden a cada una de las claves creadas por el mapper para emular todo el código de la aplicación de MapReduce.

3.5 - Prueba de la aplicación

Para ejecutar nuestra aplicación en Spark, al igual que en el caso de MapReduce, debemos empaquetarla en un archivo jar. En este caso, la distribución de Apache Spark no trae por defecto una herramienta para hacerlo, como era el caso de la distribución de Hadoop elegida anteriormente. Por lo tanto, debemos elegir una aplicación externa que lo haga. En mi caso, he seleccionado Maven debido a que es una aplicación ampliamente utilizada y sobre la que existe abundante documentación.

Una vez más, no entraré en detalles acerca del uso de esta aplicación para obtener el archivo jar necesario para ejecutar nuestra aplicación en Spark.

La prueba la he realizado sobre unos de los archivos originales de la NASA. Una parte del resultado se puede ver en la siguiente figura que sigue a estas líneas. Ni que decir tiene que he comprobado que el resultado de la aplicación de tratamiento de logs de Spark da el mismo resultado que la equivalente en MapReduce. El formato de salida es ligeramente diferente.

```
(001.msy4.communique.net,25218)
(007.thegap.com,203429)
(01-dynamic-c.rotterdam.luna.net,20903)
(01-dynamic-c.wokingham.luna.net,1121554)
(01.ts01.zircon.net.au,27063)
(02-17-05.comsvc.calpoly.edu,131166)
(02-dynamic-c.wokingham.luna.net,152676)
(023.msy4.communique.net,58811)
(03-dynamic-c.wokingham.luna.net,696320)
(0321jona.jon.rpslmc.edu,73728)
(033.msy4.communique.net,90112)
(04-dynamic-c.rotterdam.luna.net,40310)
(04-dynamic-c.wokingham.luna.net,73728)
(049.215.goodnet.com,15361)
(05-dynamic-c.rotterdam.luna.net,98304)
(05-dynamic-c.wokingham.luna.net,69067)
(052.215.goodnet.com,127045)
(06-dynamic-c.rotterdam.luna.net,65576)
(07-dynamic-c.rotterdam.luna.net,9630)
(0772jela.jelke.rpslmc.edu,1204)
(07mb369b.uni-duisburg.de,7034)
(08-dynamic-c.rotterdam.luna.net,41397)
(086.msy4.communique.net,283389)
(0875pr3e.pro.rpslmc.edu,18149)
```

Salida de SparkLog. Igual a la de HadoopLog salvo pequeñas diferencias de formato.

Aquí, al contrario a como hice con Hadoop para mostrar la diferencia entre el modo pseudo distribuido del local, no mostraré las métricas que se muestran en el gestor de datos históricos. Ya habrá tiempo para ello cuando realicemos las pruebas de las aplicaciones.

3.6 - Conclusiones

He comentado en el apartado 3.2 que la API de Spark es mucho más rica que la de MapReduce, a lo que debo añadir que es mucho más compleja. En el caso de MapReduce nos debemos adaptar a un nuevo modelo de programación, con una estructura muy específica a la hora de codificar nuestra aplicación. En Spark, sin embargo, la estructura es mucho más libre siempre y cuando nos adaptemos al uso de los RDDs. A pesar de ello, precisamos de un conocimiento mucho más profundo de la API para ver qué transformaciones podemos aplicar a cada uno de los diferentes tipos de RDD que podamos tener. Veremos esta diferencia mejor a la hora de comentar la aplicación que implementa un trabajo iterativo en Spark, aunque en la que nos ocupa ya se ve que la estructura es mucho más libre.

Otra de las cuestiones que me parece oportuno comentar es que me ha costado mucho más encontrar información sobre Spark que sobre MapReduce. Por un lado es lógico, puesto que MapReduce lleva más tiempo funcionando que Spark. Sin embargo al consultar los foros se ve que la complejidad de Spark hace que el número de personas con conocimientos profundos de Spark sea menor y la dificultad para encontrar información sobre asuntos específicos de Spark aumenta, llevando más tiempo que en el caso de MapReduce.

Por último, comentaré que el empleo de Maven me ha dado algún que otro dolor de cabeza. Principalmente debido a que desconocía dicha herramienta. Una vez que he creado mi primer artefacto y me he familiarizado con la estructura del archivo de configuración pom.xml, he reutilizado dicho artefacto para las siguientes aplicaciones, modificando manualmente los campos necesarios.

3.7 - Bibliografía

- Parsian, Mahmoud. Data Algorithms: Recipes for Scaling up with Hadoop and Spark. Sebastopol, CA: O'Reilly, 2015.

- Karau, Holden; Kowinski, Andy; Wendle, Patrick; Zaharia, Matei: Learning Spark. Lightining Fast Big Data Analysis. Sebastopol, CA: O'Reilly, 2015.

- Karau, Holden. Fast Data Processing with Spark. Birmingham, 2013: Packt.

- Sonatype. Maven, The Definitive Guide. Sebastopol, CA: O'Reilly, 2008.

- Documentación oficial de Apache Spark de la web http://spark.apache.org/

Capítulo 4

Construcción de la aplicación de cálculo iterativo para MapReduce

4.1 - Índice del capítulo 4

4.1 – Índice del capítulo 4

4.2 – Introducción

4.3 – Selección del algoritmo a implementar

4.4 – Construcción de la aplicación. Comentarios al código

4.5 – Prueba de la aplicación

4.6 – Conclusiones

4.2 - Introducción

Me enfrento ahora a un problema mucho más complejo que en el caso de construir una aplicación de tratamiento de archivos de registro para MapReduce. Al decir esto, no me refiero ni mucho menos a la complejidad matemática que pudiera tener la tarea, sino al hecho de que voy a tratar de emplear MapReduce para una tarea para la que no fue diseñado.

Como veremos, por el camino no me voy a encontrar sólo problemas técnicos intentando usar un entorno de programación para algo diferente a aquello para lo que fue diseñado, sino que me enfrento a lo que podríamos llamar "problemas de conciencia", debido a que como aspirante a arquitecto de computadoras, voy a programar una aplicación con una serie de ineficiencias graves a la hora de ejecutarse. Este aspecto lo comento en el apartado de conclusiones, y es uno de los objetivos del presente trabajo.

No he incluido un apartado dedicado a la bibliografía en este capítulo. Toda la bibliografía necesaria para este capítulo y, por ende, para el resto del trabajo, ya la he presentado en los capítulos anteriores.

4.3 – Selección del algoritmo a implementar

Para construir la aplicación objeto de esta parte del trabajo, debemos seleccionar un algoritmo iterativo que nos permita comprobar que MapReduce no es la aplicación idónea para este tipo de tareas, a la par que no nos obligue a "perdernos" en sus complejidades matemáticas.

El algoritmo que he elegido ha sido el algoritmo de Jacobi para la resolución de sistemas de n ecuaciones lineales con n incógnitas. Al profundizar en la obtención de información para la realización de trabajos iterativos para MapReduce, he descubierto que el tratamiento de matrices si bien no es eficiente, sí se adapta mejor a la estructura del modelo de programación propio de MapReduce. Además, es un algoritmo lo suficientemente simple como para no tener que dedicar un tiempo excesivo a su implementación debido a su complejidad matemática.

Aunque he comentado que la bibliografía necesaria para todo el trabajo ya se ha presentado, me veo obligado a hacer una excepción. Existe documentación más que abundante en Internet sobre este método de cómputo numérico, sin embargo me he basado en la obra:

Kreyszig, Erwin. AdvancedEngineering Mathematics, 8th Edition. Singapore, John Wiley & Sons, 1999.

Este es un libro muy querido para mí, ya que lo adquirí en un viaje de trabajo a Singapur en el año 2000. Me ha servido de consulta en muchas ocasiones a lo largo de los años, y me ha resultado entrañable el poder emplearlo en una ocasión tan importante como es este trabajo de fin de grado.

4.4 - Construcción de la aplicación. Comentarios al código

Si recordamos el modelo de programación propio de MapReduce, debemos tener presente que lo que hacemos es leer un archivo que debe estar en un formato de pares (K, V), sobre el que realizamos una serie de transformaciones en el mapper, obteniendo una salida también en formato (K, V). A su vez, esta salida nos sirve de entrada al reducer, que realizará una nueva serie de transformaciones sobre la misma, para obtener una nueva salida en formato (K, V), que se escribe en el sistema HDFS.

En muchas ocasiones, cuando empleamos MapReduce para el tratamiento de una serie de datos, no se obtiene la salida deseada en una única ejecución, sino que debemos emplear la salida de una ejecución de MapReduce como entrada de la siguiente. Proceso que deberemos ejecutar hasta obtener la salida deseada. El resultado de la una ejecución es un archivo creado en el HDFS, que se leerá en la ejecución subsiguiente. Este no es un proceso iterativo en sí mismo, sino una necesidad que se genera en el tratamiento de nuestros datos cuando usamos MapReduce o cualquier otro sistema.

En un trabajo iterativo, debemos ejecutar una determinada acción sobre unos datos de entrada, que generarán una aproximación de la solución. Para obtener una aproximación mejor, deberemos ejecutar el mismo proceso sobre la salida de la ejecución anterior y así sucesivamente, hasta realizar un número determinado de ejecuciones, o hasta que la salida de las iteraciones cumpla un determinado criterio.

En mi caso, no utilizaré criterios de cumplimiento, sino que me limitaré a realizar un número determinado de iteraciones, que se dará en forma de parámetro. Recordemos que el objeto del trabajo es determinar qué entorno de programación es mejor, por lo que nos importa menos la exactitud de la solución generada que el número de iteraciones realizadas.

Antes de empezar a comentar el código de la aplicación, debo decir que casi tan importante como el mismo es la elección de formato del archivo de entrada. En el caso que me ocupa, he necesitado dos archivos de entrada por las propias limitaciones del modelo de programación. El primero de los archivos de entrada es el sistema de ecuaciones en sí mismo. Para ello he seleccionado un sistema de tres ecuaciones con tres incógnitas, que presento a continuación. Aunque el uso de un entorno como Hadoop sería apropiado para resolver sistemas de muchas ecuaciones, almacenadas de forma distribuida, he encontrado complicado encontrar resoluciones detalladas iteración a iteración para este método que, además, convergiesen para el mismo. El sistema seleccionado estaba documentado hasta la 50ava iteración, lo cual me ha permitido comprobar en profundidad el desarrollo de la aplicación.
Este sistema lo he encontrado en la web de la universidad de Oxford:

www.robots.ox.ac.uk/~sjrob/Teaching/EngComp/linAlg34.pdf

El sistema a resolver es:

$10x - y + 2w = 6$
$-x + 11y - w + 3z = 25$
$2x - y + 10w - z = -11$
$3y - w + 8z = 15$

Mientras que la solución inicial propuesta para el sistema, sobre la que se va a empezar a iterar, es (0, 0, 0, 0).

Tenemos dos archivos de entrada, cuyos formatos son los siguientes:

system.txt	guess
	4
1 10 -1 2 0 6 25 -11 15 1	1 0
2 -1 11 -1 3 6 25 -11 15 2	2 0
3 2 -1 10 -1 6 25 -11 15 3	3 0
4 0 3 -1 8 6 25 -11 15 4	4 0

El archivo system.txt guarda los datos correspondientes al sistema de ecuaciones, empezando por el número de la fila a la que corresponde la ecuación y que va a servir de clave, mientras que guess contiene el número de ecuaciones del sistema, así como la estima inicial para resolver el sistema. En system.txt los datos están separados por espacios en blanco. No sirve otro separador. Además, debemos definir que el separador para el archivo de salida de MapReduce sea un espacio. Esto lo podemos hacer a través de los archivos de configuración o programáticamente.
Los números de columna, además de servirnos de claves para los diferentes pares (K, V), permiten saber en todo momento a la aplicación qué ecuación están tratando. Se han añadido también estos números al final para que el reducer "sepa" a qué incógnita corresponde la ecuación con la que está trabajando.

El mapper lee ambos archivos. La entrada en formato (K, V) es el sistema de ecuaciones, mientras que guess se interpreta como un archivo auxiliar. Tras manipular ambos archivos, se genera una salida en formato (K, V), cuyo destino es el reducer. El formato de esta salida es (para la primera iteración):

1 10 -1 2 0 6 25 -11 15 0 0 0 0 1
2 -1 11 -1 3 6 25 -11 15 0 0 0 0 2
3 2 -1 10 -1 6 25 -11 15 0 0 0 0 3
4 0 3 -1 8 6 25 -11 15 4 0 0 0 0 4

recibiendo cada reducer una de las ecuaciones y, por lo tanto, calculando la estima correspondiente al resultado de la variable correspondiente a su fila.

El formato puede parecer redundante, y podría parecer suficiente emplear tan solo un archivo de entrada con toda la información. Sin embargo, para calcular la estima de cada una de las variables necesitamos las estimas de todas ellas de la iteración precedente, así como el vector de resultados de las ecuaciones completo. Además, cada reducer va a escribir sólo la salida correspondiente a la ecuación que ha recibido, y en la siguiente iteración precisa nuevamente la información correspondiente a las tres estimas.
Se debe tener en cuenta que cada iteración es un trabajo diferenciado de MapReduce, y no guardamos nada en memoria. Los diferentes trabajos se deben comunicar a través del HDFS.

La decisión de añadir el número de ecuaciones en el archivo guess pretende eliminar un acceso más al HDFS. Mientras que el mapper va a leer todas las líneas del archivo system.txt y no necesita esta información, el reducer necesita saber el número de ecuaciones para calcular las diferentes estimas. Podría haber implementado un contador de las líneas del archivo pero precisaría de un acceso más al sistema de archivos, lo que haría aún más ineficiente el programa.

Un problema con el que nos encontramos es que el reducer debe proporcionar una salida en formato (K, V), exigencia del modelo de programación de MapReduce. Lo que he hecho ha sido crear un archivo temporal llamado TMP_Iteration, donde se van creando diferentes subdirectorios en los que se almacena esta salida, que no es otra que el propio sistema de ecuaciones propiamente dicho. Una vez terminada la ejecución, este directorio se borra, puesto que la salida que proporciona ya está disponible en el directorio de entrada, donde se encuentra el archivo system.txt. La salida realmente útil que genera el reducer es un nuevo archivo guess que, junto con el archivo system.txt original, va a ser la entrada necesaria para la nueva iteración.

4.5 - Prueba de la aplicación

La prueba de la aplicación es directa. Tras compilarla y empaquetarla en formato jar con el nombre iter.jar, ejecutamos el siguiente comando:

```
$ bin/hadoop jar iter.jar HadoopIter /input n
```

Los parámetros son "/input", que es el directorio donde se encuentra el archive system.txt, y "n", que es el número de iteraciones deseado. Para esta prueba he seleccionado n = 10 y el resultado es el siguiente:

```
4
1 1.0001186
2 1.999768
3 -0.99982816
4 0.99978596
```

La solución de este sistema es (1, 2, -1, 1), por lo que el resultado es correcto. Durante la depuración de la aplicación he mantenido el directorio TMP_Iteration con todos sus subdirectorios, de modo que he podido comprobar que los resultados de las diferentes iteraciones eran correctos.

4.6 - Conclusiones

La problemática más evidente e importante es la obligación, impuesta por el modelo de programación de MapReduce, de emplear un archivo de entrada y otro de salida para cada tarea. Además, debemos recordar que dichos archivos están almacenados en un sistema

de archivos distribuido, por lo que es costoso acceder a ellos. Este hecho genera ineficiencias en el sistema, ya que cada iteración debe efectuar dos accesos al HDFS de forma obligatoria.

Por si fuera poco, la necesidad de crear un archivo que comunique cada una de las tareas (el guess) genera un mayor número de ineficiencias en la ejecución, al tener que perderse más tiempo accediendo al sistema de archivos.

Hoy por hoy, existen métodos dentro del ecosistema de Hadoop para sortear estas ineficiencias y comunicar las diferentes tareas de forma menos costosa, pero he querido mostrar una ejecución MapReduce pura, tal y como se pensó en el sistema inicial. Hay que tener siempre en mente que los métodos iterativos no eran el objetivo de dicho sistema inicial.

Como dificultad añadida, aunque no tiene nada que ver con el modelo de programación de MapReduce, ha sido el encontrar sistema de ecuaciones convergentes para el método de Jacobi. He probado con diferentes sistemas, tanto hallados en la bibliografía como creados por mí de forma aleatoria, que divergían rápidamente al usar el método de Jacobi (las estimas se iban al orden de 10^{32} en unas pocas iteraciones).
El método de Gauss-Seidel está mucho más documentado, ya que es mucho más utilizado al ser más eficiente. Sin embargo no lo he seleccionado ya que su implementación hubiera sido más compleja y lo que pretendía era ejemplificar las ineficiencias de MapReduce en los trabajos iterativos.

A nivel más personal diré que me ha resultado muy tedioso el tratamiento de cadenas, así como el trabajo con arrays debido a la necesidad de ser cuidadoso con los índices de los mismos. Sin embargo, el tratamiento de cadenas puede decirse que es una característica inherente al trabajo con MapReduce, debido a la propia naturaleza del modelo de programación.

Para terminar, podemos ver que he proporcionado dos archivos de código fuente para esta aplicación; uno es el correspondiente a la ejecución en modo local y otro en modo semi distribuido. La diferencia entre ambos no es grande, limitándose al modo de escribir en el archivo guess. Para el caso local empleo la clase BufferedWriter, mientras que para el caso semi distribuido empleo la clase FSDataOutputStream. Los métodos para efectuar la escritura se denominan write() en ambos casos y son muy similares.

Capítulo 5

Construcción de la aplicación de cálculo iterativo para Spark

5.1 - Índice del capítulo 5

5.1 – Índice del capítulo 5

5.2 – Introducción

5.3 – Construcción de la aplicación. Comentarios al código

5.4 – Prueba de la aplicación

5.5 – Conclusiones

5.2 - Introducción

Como última tarea antes de comenzar la construcción de la aplicación de gestión automatizada de pruebas en los entornos de MapReduce y Spark, debo construir la aplicación de cálculo iterativo para Spark. Esta aplicación debe dar un resultado equivalente al de la aplicación del capítulo anterior,

Como ya hemos comentado a lo largo de la memoria, Spark está pensado para hacer un uso intensivo de la memoria, por lo que deberé crear una aplicación que maximice el uso de la memoria en lugar de los accesos al HDFS, lo cual le permita incrementar su velocidad de ejecución. En teoría, como ya he comentado anteriormente, el incremento en la velocidad de ejecución es de un orden de magnitud.

Aunque, como ya veremos, Spark nos permite una mayor libertad de acción como modelo de programación, con unas APIs más ricas, dichas APIs siguen siendo limitadas y debemos condicionar nuestro código a dichas limitaciones para que el funcionamiento de nuestra aplicación sea óptimo. Esta afirmación no deja de ser una verdad de Perogrullo, puesto que es aplicable a cualquier modelo de programación, pero es te hecho me ha dado numerosos quebraderos de cabeza y generado una serie de problemas que he tenido que ir resolviendo poco a poco.

Pasemos a los comentarios al código, donde se puede ver este hecho.

5.3 - Construcción de la aplicación. Comentarios al código

Ni que decir tiene que el algoritmo iterativo a implementar es el mismo que en el caso de MapReduce, el algoritmo de Jacobi para la resolución de un sistema de n ecuaciones con n incógnitas.

En este caso, la entrada será un único archivo, denominado system.txt, que tendrá el formato siguiente:

1 4 2 3 8 -14 27 1
2 3 -5 2 8 -14 27 2
3 -2 3 8 8 -14 27 3

Podemos distinguir en este formato el mismo sistema de ecuaciones que en el caso de MapReduce. Lo reproduzco a continuación para una mejor lectura:

$4x + 2y + 3z = 8$
$3x – 5y +2z = -14$
$-2x + 3y + 8z = 27$

Si recordamos el caso de MapReduce, teníamos un segundo archivo donde se localizaba el vector de valores estimados para el sistema de ecuaciones. En el caso de Spark se sustituye

por un array de floats. De hecho, en el código podemos ver dos definiciones de variables seguidas:

Final Float [] estimatedValues = new Float [] {0.0f, 0.0f, 0.0f};
Final Integer variable_number = 3;

Por un lado, definimos el vector de valores estimados para nuestra solución, mientras que por otro le decimos al programa el número de ecuaciones que tiene el sistema. En principio pensé en que dicho número se pasase como un parámetro al programa pero, por la propia estructura de la aplicación, me complicaba el paso de parámetros y lo he dejado de esta forma. La aplicación se tiene que compilar para cada caso diferente, sin embargo como mi intención es la de acceder lo menos posible al HDFS y el vector de valores estimados de la solución lo voy a tener definido en el código, ya es necesario compilar la aplicación para cada sistema nuevo. El tiempo de compilación es de 3 segundos en mi ordenador de sobremesa, por lo que no creo que sea un problema el tener que hacerlo para cada nuevo sistema de ecuaciones.

Al final de cada línea vemos que tenemos el número de fila de la matriz de coeficientes. El comportamiento de los RDDs dentro del entorno de Spark es algo sobre lo que no tenemos control, y la aplicación necesita saber en todo momento cuál es la ecuación con la que está trabajando.

La decisión de definir en el código el vector de valores estimados se debió no solo a tratar de emplear la característica de Spark para usar al máximo la memoria, sino porque si trataba de copiar la estructura de la aplicación de MapReduce, y creaba dos RDDs diferentes, uno para las ecuaciones y otro para las estimas, el limitado número de operaciones que proporciona la API de Spark para combinar dos RDDs me impedía elaborar correctamente la aplicación.

Podemos ver en el código que lo primero que hacemos es cargar el archivo de las ecuaciones (sobre un RDD) y mapearlo a un JavaPairRDD. Los PairRDDs me resultan convenientes puesto que la estructura de los mismos se conserva tras aplicar las diferentes transformaciones sobre ellos, cosa que no ocurre con un RDD. Este hecho me resulta útil de cara a que el orden de las ecuaciones se conserve en todo momento en los RDDs que se vayan creando, aunque tampoco tiene por qué ser estrictamente necesario.

En este primer PairRDD simplemente define los pares (K, V) del mismo. Posteriormente lo mapeamos a un nuevo PairRDD sobre el que se efectúan las iteraciones. Aunque podría haber creado un único PairRDD sobre el que realizar todo el trabajo, esta forma es más clara y no sobrecarga excesivamente las operaciones a realizar sobre el HDFS comparado con las operaciones que van a generar las iteraciones.

La parte más interesante del código, y la que más dolores de cabeza me dio es, precisamente, la definición de este segundo PairRDD. Para conseguirlo consulte la página de ejemplos de Apache Spark en la web:

http://spark.apache.org/examples.html

concretamente, el ejemplo de regresión logística. Dicho ejemplo tiene un error, ya que podemos ver que se emplea la palabra reservada **extends** a la hora de definir la clase anónima, pero debemos emplear la palabra reservada **implements**, ya que lo que hacemos es implementar la interfaz Function de Spark para simular el paso de funciones en Java.

Como ya he dicho, el vector de valores estimados está definido en un array de floats. Esta variable está en memoria local y el comportamiento de una variable de este tipo dentro del

clúster es indeterminado. Sin embargo, lo que he hecho ha sido separar la definición de la implementación de la clase anónima, y proporcionarle un constructor en el que necesitamos como parámetro el vector de valores estimados. De este modo, y como se indica en la página web que cito, en cada iteración se pasa el nuevo valor del vector a todo el clúster por lo que, en todo momento, las diferentes copias locales del vector de valores estimados están en un estado consistente. En concreto el valor del vector se pasa en cada llamada a la transformación mapValues().

Al igual que en el caso de la aplicación para MapReduce, he comprobado la validez de la implementación hasta la 50ava iteración, con resultado correcto.
No se genera un archivo de salida, sino que las diferentes aproximaciones se van presentando por pantalla. Quiero remarcar con ello que en Spark no es obligatorio generar un archivo de salida, como sucedía con MapReduce. Aunque es lógico que se genere dicho archivo de salida, ello nos genera un coste al tener que escribir sobre el HDFS, que en el caso de Spark nos podemos ahorrar.

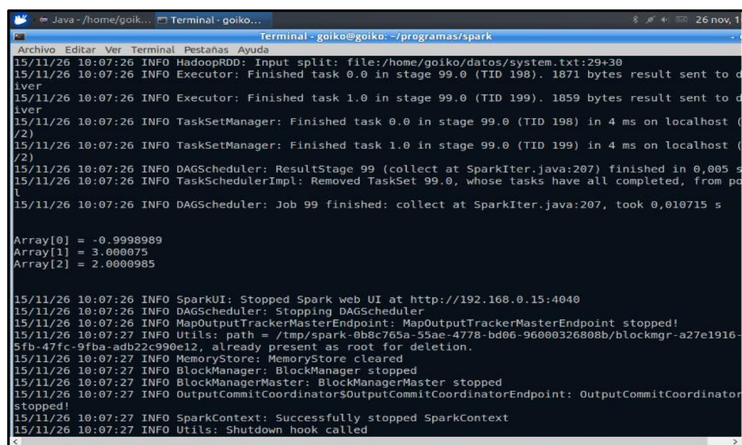
5.4 - Prueba de la aplicación

La prueba no presenta tampoco ninguna complicación. Tras compilarla y empaquetarla en formato jar con Maven, ejecutamos el comando siguiente en la consola:

$ bin/spark-submit –class SparkIter /directorio_archivo_jar/SparkIter-v1.3.jar n

donde el parámetro n es el número de iteraciones deseado.

Si ejecuto con n = 50, y tras un tiempo de espera sensiblemente inferior al de la ejecución equivalente en MapReduce, obtengo la salida siguiente:



Salida de SparkIter para un sistema de 3 ecuaciones con 3 incógnitas.

Es un resultado correcto para el sistema planteado con el número de iteraciones indicado.

5.5 - Conclusiones

Se puede ver que la extensión del apartado de comentarios al código es superior a otros casos. Esta ha sido la aplicación más compleja de construir debido, sobre todo, a la complejidad de la API de Spark, aunque también a la complejidad del código Java que he empleado en la misma. La construcción de la aplicación equivalente para MapReduce también tuvo su importancia, pero más debido al empleo del sistema de archivos que a una complejidad implícita a la propia aplicación.

Hemos visto también que el empleo de variables en Spark no debe tomarse a la ligera. Debemos tener presente en todo momento que estamos trabajando en un sistema de archivos distribuido, y que el comportamiento de una variable definida localmente es indeterminado. El artificio empleado en la aplicación es una forma de asegurarnos de que las variables que tenemos en memoria se envían a todo el clúster en el momento en el que van a ser empleadas.

Aunque aún no me he preocupado de las métricas de ejecución, al ejecutar la aplicación construida en este apartado la sensación es que tarda mucho menos que la equivalente para MapReduce. Las pruebas las he realizado localmente, y no tienen por qué ser extrapolables al clúster, pero el menor tiempo de ejecución se debe a un mayor empleo de la memoria en detrimento del sistema de archivos. Esta es una característica de Spark, y es de esperar que su comportamiento en modo distribuido sea el mismo en lo que a tiempo de ejecución se refiere.

Capítulo 6

Generadores de tráfico

6.1 - Índice del capítulo 6

6.1 – Índice del capítulo 6

6.2 – Introducción

6.3 – Generadores de tráfico basados en HadoopLog y SparkLog

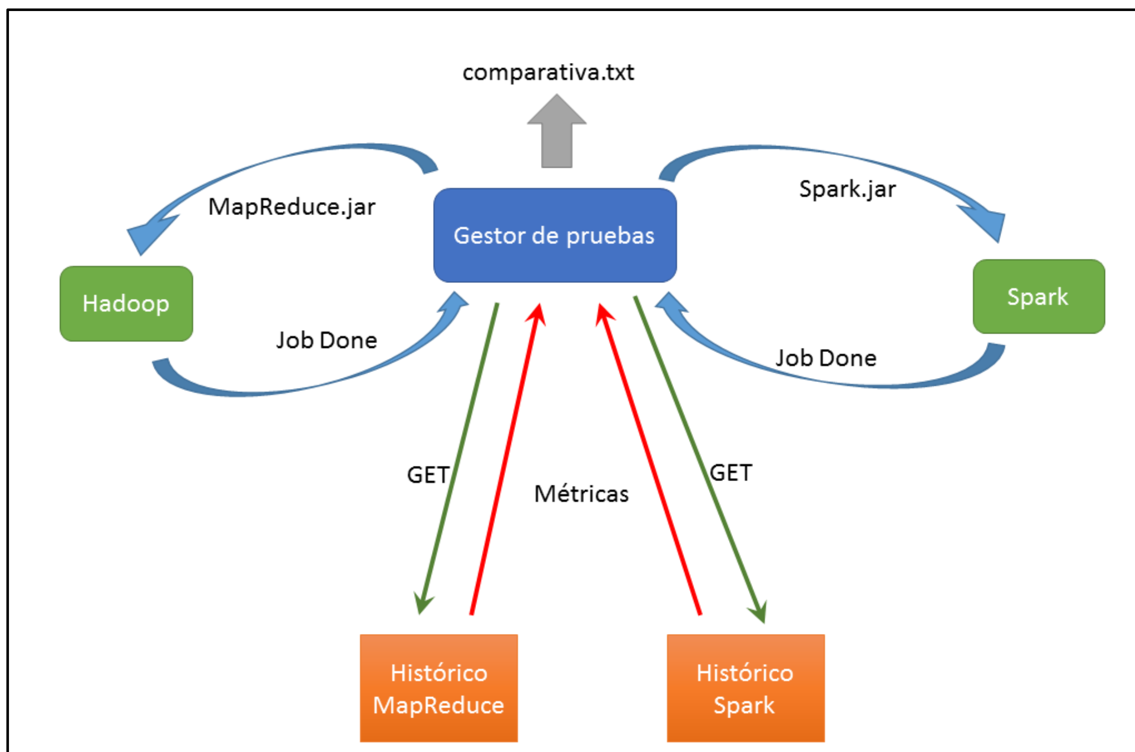
6.4 – Generador de tráfico genérico

6.5 – Conclusiones

6.2 - Introducción

Llegados a este punto, nos planteamos la posibilidad de crear un generador de tráfico que enviase trabajos tanto a MapReduce como a Spark y que nos devolviese un resumen de las diferentes pruebas realizadas, con las métricas más relevantes de las mismas.

El objetivo era una aplicación que enviase los archivos jar, primero a MapReduce y después a Spark, tanto de las aplicaciones de procesamiento de archivos de registro como de las aplicaciones de cálculo iterativo y generase un informe con las métricas del que pudiésemos sacar conclusiones significativas. Las métricas las debería obtener de los servidores de históricos de MapReduce y Spark. La arquitectura debería ser la siguiente:



Arquitectura inicial para el generador de tráfico

Tras estudiar la posible implementación de dicha aplicación, me encontré con una serie de problemas que me hicieron desistir de realizar una aplicación válida para ambos entornos de programación, y construir una para cada uno de ellos.
Uno de los problemas era la dificultad para obtener la identidad de los trabajos ejecutados en ambos entornos desde una aplicación externa. Veremos más adelante también que, mientras que en una aplicación MapReduce es trivial el obtener dicha identidad, no lo es tanto en Spark. Podríamos hacer que las identidades de los diferentes trabajos ejecutados en cada uno de los entornos se guardasen en un archivo, que luego leyese el generador de tráfico, pero era reacio a emplear el sistema de archivos en el entorno distribuido debido a las latencias, aunque en el caso de una aplicación de pruebas de rendimiento, el rendimiento no es un factor crítico. Para el caso del generador de tráfico de Spark lo que hice finalmente fue extraer desde el servidor de históricos las identidades de los últimos trabajos ejecutados, como se verá en el apartado correspondiente.

Esta solución podría haberse empleado finalmente en un generador de tráfico genérico para ambos entornos y, aunque fue un motivo de peso para la realización de varios generadores de tráfico al principio, según avancé en el trabajo no fue la razón determinante al hallar la solución descrita. El motivo principal para la construcción de varios generadores de tráfico fue que las métricas disponibles para MapReduce y Spark desde los servidores de históricos eran muy diferentes y no permiten una comparación directa más allá del tiempo total de ejecución. Hubiera sido realmente interesante poder comparar el uso del HDFS y de la memoria para ejecuciones equivalentes en ambos entornos.

Me extenderé un poco en el tema de las métricas porque, por un lado creo necesario explicar la decisión de no construir un generador genérico, y por otro porque me parece interesante.
Tanto MapReduce como Spark ofrecen un servidor de históricos que nos permiten estudiar los parámetros de configuración y de ejecución de los diferentes trabajos ejecutados. En ambos casos están desactivados por defecto y debemos arrancarlos de forma explícita. En el caso de MapReduce debemos ejecutar:

$ sbin/mr-jobhistory-daemon.sh start historyserver

mientras que en Spark:

$ sbin/start-history-server.sh

Una vez arrancados, tenemos sendos UIs accesibles a través del navegador. Para MapReduce en **localhost:19888** y para Spark en **localhost:18080**. Para ejecuciones en clúster, la dirección y el puerto cambian y se definen en los archivos de configuración. En estas direcciones, y en los directorios que podemos ver en la documentación oficial, tenemos disponibles los archivos en formatos JSON (o XML, CSV y algún otro) con las métricas de las ejecuciones. En otro servidor tenemos los trabajos ejecutándose en un preciso momento, pero puesto que mis ejecuciones son cortas, no merece la pena meterse con ellos.

Para un trabajo concreto ya finalizado, el aspecto del UI de MapReduce es:

Vista de los datos ofrecidos por el servidor de históricos en el UI de MapReduce.

Dentro del apartado "Counters" podemos ver las diferentes métricas de la ejecución de este trabajo. Mostrando una pequeña parte podemos ver lo siguiente:

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	221.230	22.660	243.890
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	8.750	1.347	10.097
	Input split bytes	1.212	0	1.212
	Map input records	13.846.448	0	13.846.448
	Map output bytes	322.827.108	0	322.827.108
	Map output materialized bytes	350.520.076	0	350.520.076
	Map output records	13.846.448	0	13.846.448
	Merged Map outputs	0	12	12
	Physical memory (bytes) snapshot	3.221.508.096	307.752.960	3.529.261.056
	Reduce input groups	0	137.978	137.978
	Reduce input records	0	13.846.448	13.846.448
	Reduce output records	0	137.978	137.978
	Reduce shuffle bytes	0	350.520.076	350.520.076
	Shuffled Maps	0	12	12
	Spilled Records	13.846.448	13.846.448	27.692.896
	Total committed heap usage (bytes)	2.468.872.192	207.618.048	2.676.490.240
	Virtual memory (bytes) snapshot	8.630.800.384	740.352.000	9.371.152.384
	Name	Map	Reduce	Total

Detalle de los contadores de una ejecución de MapReduce en el UI

Vemos que tenemos una serie de métricas divididas en totales, operaciones "map" y operaciones "reduce". Muestro esta parte porque tenemos las métricas de tiempo de CPU y la cantidad de registros escritos en bytes.

Si mostramos lo que nos ofrece el servidor de históricos en Spark vemos los diferentes trabajos ejecutados hasta el momento:

History Server - Mozilla Firefox

localhost:18080

Spark 1.5.1 History Server

Event log directory: file:/tmp/spark-events

Showing 1-6 of 6

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
app-20151210184803-0004	SparkIter	2015/12/10 18:48:00	2015/12/10 18:48:08	7 s	goiko	2015/12/10 18:48:08
app-20151210184245-0003	SparkIter	2015/12/10 18:42:43	2015/12/10 18:42:51	8 s	goiko	2015/12/10 18:42:51
app-20151210184149-0002	SparkIter	2015/12/10 18:41:47	2015/12/10 18:41:55	8 s	goiko	2015/12/10 18:41:55
app-20151210183917-0001	SparkIter	2015/12/10 18:39:15	2015/12/10 18:39:21	6 s	goiko	2015/12/10 18:39:21
app-20151210183857-0000	SparkIter	2015/12/10 18:38:55	2015/12/10 18:39:01	7 s	goiko	2015/12/10 18:39:01
local-1449768959764	SparkIter	2015/12/10 18:35:57	2015/12/10 18:36:02	5 s	goiko	2015/12/10 18:36:02

Show incomplete applications

Detalle del servidor de históricos de Spark tal y como se muestra en el UI.

Pero si seleccionamos un trabajo concreto:

SparkIter (app-20151210191832-0005) - Spark Jobs - Mozilla Firefox

localhost:18080/history/app-20151210191832-0005/jobs/

Spark 1.5.1 — Jobs · Stages · Storage · Environment · Executors — SparkIter (app-20151210191832-0005) application UI

Spark Jobs (?)

Scheduling Mode: FIFO
Completed Jobs: 8

Event Timeline

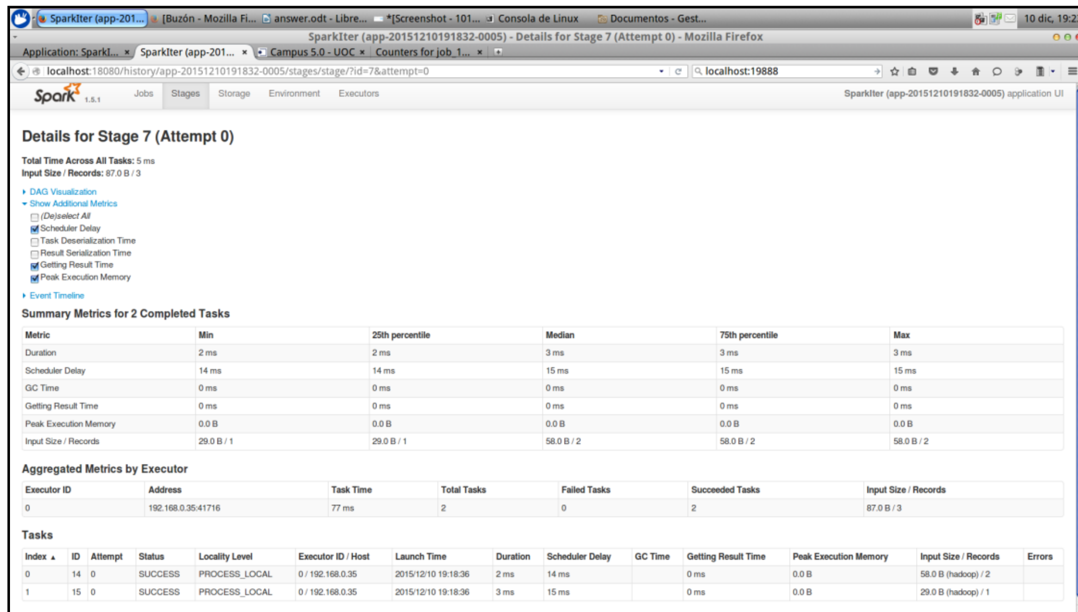
Completed Jobs (8)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	collect at SparkIter.java:207	2015/12/10 19:18:36	50 ms	1/1	2/2
6	collect at SparkIter.java:207	2015/12/10 19:18:36	52 ms	1/1	2/2
5	collect at SparkIter.java:207	2015/12/10 19:18:36	46 ms	1/1	2/2
4	collect at SparkIter.java:207	2015/12/10 19:18:36	47 ms	1/1	2/2
3	collect at SparkIter.java:207	2015/12/10 19:18:36	46 ms	1/1	2/2
2	collect at SparkIter.java:207	2015/12/10 19:18:36	46 ms	1/1	2/2
1	collect at SparkIter.java:207	2015/12/10 19:18:36	47 ms	1/1	2/2
0	collect at SparkIter.java:207	2015/12/10 19:18:34	2 s	1/1	2/2

Detalle de las tareas en las que se subdivide un trabajo de Spark.

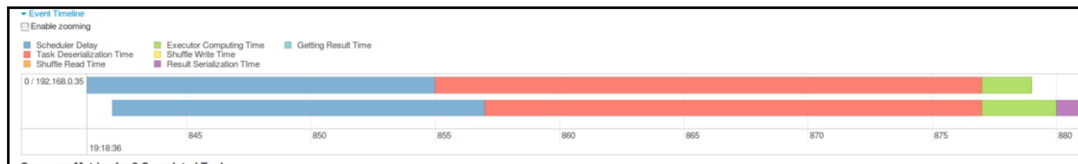
Vemos las diferentes tareas en las que se ha subdividido dicho trabajo, y las métricas que podemos obtener son para cada una de las tareas por separado. Si pensamos que, para la aplicación de cálculo iterativo tenemos 1 ó 2 tareas por iteración, y que he probado la aplicación hasta con 1000 iteraciones en Spark, nos podemos hacer una idea de la dificultad de la extracción de las métricas. Aun tomándonos

dicho trabajo o automatizándolo, las métricas disponibles son muy diferentes a las que obtenemos en MapReduce, como podemos ver en la figura siguiente:



Etapa de una tarea de Spark.

Una de las partes que me parece más interesante de estas métricas es que haciendo click en **Event Timeline** obtenemos un gráfico como el siguiente:



Línea de tiempo de una etapa de Spark donde se ve el tiempo dedicado a la planificación.

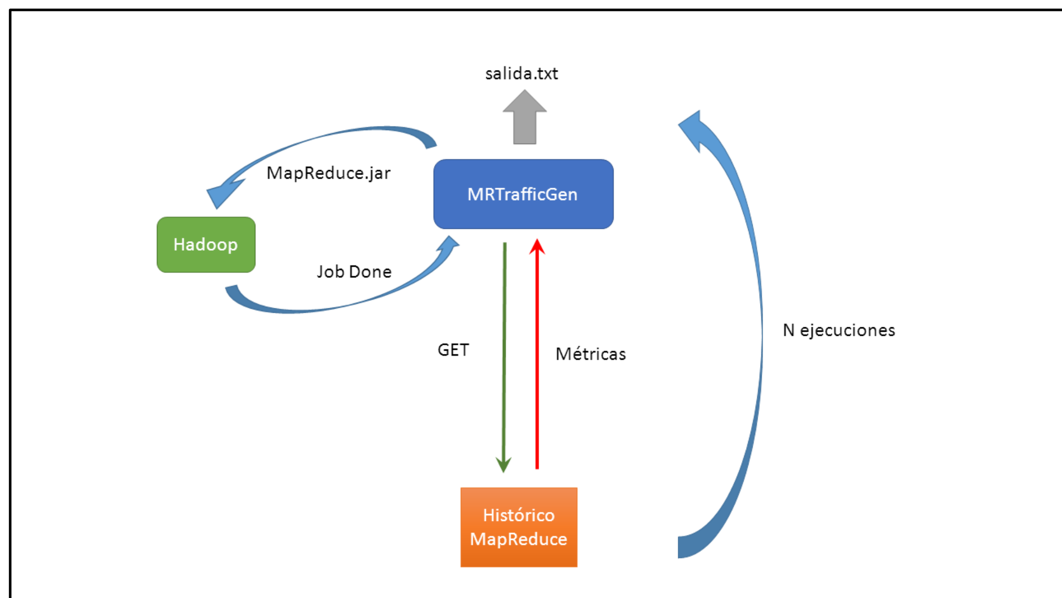
En el podemos ver, entre otras cosas, el overhead debido a la planificación de las diferentes tareas, que es uno de los parámetros más importantes en un sistema de computación distribuida.

Otra de las decisiones a tomar fue la del formato de archivo que los generadores de tráfico iban a leer de los históricos. Mientras que MapReduce nos ofrece tanto XML como JSON, Spark tan sólo nos permite leer el formato JSON, por lo que me decidí por dicho formato.
Es un formato conocido, empleado por un gran número de aplicaciones y para el que existen abundantes bibliotecas para Java muy documentadas. Aunque se creó inicialmente para JavaScript, se emplea actualmente en aplicaciones escritas en numerosos lenguajes de programación.

6.3 – Generadores de tráfico basados en HadoopLog y SparkLog

Una vez tomada la decisión de generar tráfico de forma independiente para MapReduce y Spark, era trivial pensar que los generadores debían ser aplicaciones específicas para cada uno de los entornos, y que iban a estar basadas en las ya construidas.

Para el caso de MapReduce empecé con la aplicación HadoopLog y comencé a modificar la clase Driver para que crease diferentes configuraciones y fuese ejecutando los diferentes trabajos en las mismas. La arquitectura de la aplicación final es la siguiente:



Arquitectura final del generador de tráfico para MapReduce

La idea es que vamos a ejecutar cierto número de trabajos sobre un idéntico número de archivos de registro. La estructura de dichos archivos de registro deberá ser idéntica, pero deberán tener un número diferente de líneas, es decir, vamos a ir variando el tamaño del problema y vamos a ir registrando lo que sucede.

La aplicación, denominada **MRGen**, precisa de tres parámetros; nombre genérico de los directorios de entrada, nombre genérico de los directorios de salida y número de ejecuciones. Si vamos a realizar tres ejecuciones, deberemos crear tres directorios de entrada. Si el nombre genérico pasado como parámetro es **input**, los directorios serán **input0**, **input1** e **input2**. Si el nombre genérico de salida es **output**, los directorios en los que se escribirá las diferentes salidas serán **output0**, **output1** y **output2**. El tercer parámetro será, evidentemente, 3.

El driver ejecuta un bucle con el número de ejecuciones y va creando diferentes configuraciones del entorno Hadoop en las que va ejecutando los trabajos. Al inicio de cada trabajo se obtiene la identidad del mismo con la instrucción:

```
String jobid = job.getJobID().toString();
```

y al finalizar el mismo, se obtienen las métricas correspondientes a ese trabajo y se añaden a un archivo de texto de salida.

La aplicación en sí es básicamente la misma que la construida en el capítulo 2, siendo la gran diferencia el acceso al servidor de históricos y el tratamiento de los archivos JSON. Para dicho tratamiento se ha empleado la biblioteca **javax.json**, pudiéndose ver en el código de forma bastante clara el mismo. Para dar una explicación exhaustiva tendría que dar un ejemplo del archivo JSON obtenido del servidor, pero sería bastante engorroso. Creo que basta decir que se van obteniendo los diferentes JsonObjects y JsonArrays de los que a su vez obtenemos los JsonValues correspondientes a las métricas que queremos extraer.
Si bien podría haber empleado un parser, me ha resultado muy instructivo de cara a la comprensión de la estructura del formato JSON el hacerlo directamente. Aunque el concepto de dicho formato es simple, la posibilidad de ir anidando objetos y arreglos hace que los archivos lleguen a tener complejidades interesantes.

Una vez terminada esta aplicación, y tras las pruebas de funcionamiento correspondientes, en lugar de continuar con MapReduce decidí continuar con una aplicación para Spark basada en SparkLog. Por un lado porque tenía frescos los detalles de la aplicación de tratamiento de logs, y por otro porque empezaba a estar justo de tiempo y quería al menos tener un generador de tráfico funcional para cada entorno de programación en caso de no tener tiempo para más.

Esta aplicación, como sucedía con la anterior, está basada en SparkLog y se denomina SparkTrafficGen. En este caso necesitamos los mismos parámetros que en MRGen, generándose los mismos directorios de salida. También, como en el caso de MRGen, se crea un bucle que va creando las diferentes configuraciones y contextos necesarios para enviar las diferentes ejecuciones. La arquitectura de la aplicación es prácticamente la misma que la mostrada en la figura para MRGen.

Como ya he comentado por encima en la introducción, uno de los problemas fue obtener la identidad del trabajo ejecutado. En mis dos aplicaciones de Spark anteriores empleé la clase **JavaSparkContext** para crear el contexto donde se ejecuta la aplicación. En mi inexperiencia en este entorno, pensé que las clases de Scala se renombraban en Java con la palabra Java por delante, como es el caso de JavaPairRDD y otras. Sin embargo, en Java también tenemos disponible la clase **SparkContext**, cuyo funcionamiento y cometido es muy similar al de la anterior. Una de las diferencias entre ambas es que la clase SparkContext implementa el método **applicationId()**, cosa que no hace JavaSpakContext, y que devuelve la identidad de la aplicación que se está ejecutando en dicho contexto.
Cuando me di cuenta traté de modificar el código para emplear SparkContext en lugar de JavaSparkContext, pero necesitaba reescribir prácticamente toda la aplicación.

Finalmente, lo que hace la aplicación es esperar a que el servidor de históricos reciba los datos de la última ejecución y obtiene la identidad del archivo JSON correspondiente. En el momento de escribir estas líneas la espera está configurada a 10 segundos un poco “a ojo”. En un entorno distribuido dicha espera sería diferente (seguramente mayor) y habría que modificarla o variar el sistema para obtener esta identidad.

Posteriormente se leen las métricas correspondientes y se guardan en un archivo de texto.

También en el momento de escribir estas líneas el control de excepciones deja bastante que desear. Me he encontrado con otros problemas en este aplicación y la he estado escribiendo un poco a salto de mata. Prefiero continuar y mejorarla más adelante si hay tiempo que demorarme en detalles que puedan comprometer la entrega del trabajo.

En esta aplicación también empleo la biblioteca **javax.json** para el tratamiento de los archivos JSON obtenidos del servidor de históricos. Dicha biblioteca me ha proporcionado bastante entretenimiento debido, en gran medida, a un pequeño despiste causado en gran medida por lo novedosas que me resultan la mayor parte de las tecnologías que estoy empleando en el proyecto.

Para MapReduce tenemos la variable de entorno **HADOOP_CLASSPATH** que, como podemos adivinar por su nombre, le indica a Hadoop dónde buscar las clases del proyecto. La necesidad de dicha variable se debe a que estamos trabajando en un entorno distribuido y mientras que la variable **CLASSPATH** permite a Java encontrar las clases en tiempo de compilación, estas clases no están disponibles en tiempo de ejecución en los nodos de Hadoop. El lugar donde se encuentran las clases que deben de estar disponibles en tiempo de ejecución se encuentra definido en **HADOOP_CLASSPATH**, y es el propio Hadoop quien se encarga de gestionar esta disponibilidad para todos los nodos.

En el caso de Spark no tenemos dicha variable, por lo que obtuve excepciones

```
Exception in thread "main" java.lang.NoClassDefFoundError
```

en tiempo de ejecución para las clases contenidas en javax.json (error de novato cuando se empiezan a manejar bibliotecas externas). El lugar donde se encuentran las clases que Spark debe proporcionar a todos los nodos del entorno en tiempo de ejecución se debe definir en el script de configuración **spark-env.sh**, que se encuentra en el directorio **conf** de la instalación. Inicialmente no tenemos dicho script, sino el archivo **spark-env.sh.template** donde se nos da un esqueleto del mismo e indicaciones para su elaboración. El estudio de esta plantilla es útil de cara a conocer qué podemos modificar del entorno de Spark para la correcta (o más eficiente) ejecución de nuestra aplicación.

6.4 – Generador de tráfico genérico

Ya comenté en el apartado 2 de este capítulo que desistí de hacer un lanzador genérico de pruebas para ambos entornos, y explicaba los motivos. Sin embargo, llegado a este punto pensé que podía ser interesante el tratar de construir un lanzador de pruebas externo, aprovechando el sistema de leer la identidad del último trabajo desde el servidor de históricos.

Aunque en el título denomino esta aplicación como generador de tráfico genérico, la aplicación está escrita para HadoopIter aunque, como se puede ver, podríamos adaptarla fácilmente a cualquier archivo jar de cualquier aplicación para MapReduce o Spark que creemos.

La aplicación la he denominado MRTrafGen, para diferenciarla de MRGen. Como parámetros necesita un nombre de directorio genérico de entrada, un nombre de directorio genérico de salida, el número de trabajos totales a ser generados y el número de iteraciones que deben llevar a cabo los trabajos. Esta estructura de parámetros es similar a la descrita para MRGen, aunque lo que debemos tener en cada uno de los directorios es diferente. En los directorios de entrada debemos tener los archivos JAR que van a enviarse a Hadoop. La

aplicación va recorriendo los diferentes directorios de entrada, lee el archivo JAR contenido en cada uno de ellos y lo envía a Hadoop con el parámetro de iteraciones introducido. Hay que tener en cuenta que también debemos enviar el nombre de la clase principal contenida en dicho archivo JAR, por lo que debe ser la misma en todos los casos.

Un defecto que tiene esta aplicación, al menos a la hora de escribir estos comentarios, es que el número de iteraciones es el mismo para todas las ejecuciones. Se podría modificar la aplicación para que admitiese un número de parámetros de iteraciones igual al número de ejecuciones, de modo que cada ejecución tuviese un número determinado de iteraciones diferente. Esto permitiría jugar no sólo con unos archivos JAR diferentes (sistemas de ecuaciones distintos), sino que podríamos ejecutar diferentes números de iteraciones para un mismo sistema, lo que nos daría una mayor número de combinaciones posibles a la hora de realiza los experimentos.

Una vez lanzada cada una de las ejecuciones, la aplicación espera a que el servidor obtenga los datos del último trabajo, lee la identidad de dicho trabajo del servidor de un archivo en formato JSON y obtiene las métricas correspondientes a dicho trabajo, también desde un archivo en formato JSON.
La mecánica de trabajo para estas acciones es idéntica a la de la aplicación generadora de tráfico SparkTrafficGen.

La construcción de esta aplicación no ha presentado grandes problemas en sí misma, ya que, por un lado es una aplicación Java genérica y, por otro, aprovechaba código ya escrito para otras aplicaciones. En el apartado de Java puro, podemos destacar el uso de las clases **ProcessBuilder** y **Process** para ejecutar una aplicación externa. La clase ProcessBuilder nos permite configurar la aplicación que se va a ejecutar, incluyendo la línea de comandos necesaria, mientras que el objeto Process, que es la aplicación en sí, la devuelve la llamada al método **start()** de la clase ProcessBuilder.

Una vez escrita la primera versión de esta aplicación, que consistía en un programa en Java que se limitaba a llamar a Hadoop, me encontré con algunos problemas que estuvieron a punto de hacerme desistir de realizar esta aplicación en Java. A partir de un cierto número de iteraciones, la ejecución de la aplicación se bloqueaba y no volvía a reanudarse. Tras construir una aplicación análoga en C++ y realizar la llamada a la aplicación de Hadoop no me encontraba con dichos problemas, por lo que me planteé el realizar el generador de tráfico genérico en este lenguaje de programación. La mayor complejidad de realizar las peticiones GET al servidor de históricos en este lenguaje, algo en lo que no tenía experiencia, y el deseo de aprovechar la parte de código ya escrita en Java a tal efecto me hicieron investigar más al respecto de este error en la ejecución.
El problema residía en que la clase ProcessBuilder lleva asociados unos buffers de entrada y salida para llevar a cabo la ejecución del proceso externo. Para una salida tan "verbosa" como es la de Hadoop (aunque se puede configurar para que no lo sea tanto), los buffers se saturaban rápidamente para un número de iteraciones tan pequeño como 10. La solución consiste en redireccionar dichos buffers a archivos, de modo que se vayan drenando en dichos archivos y no se saturen. La parte del código que lleva a cabo esta labor es:

```
File output = new File("./output.txt");

File err = new File("./errors.txt");

pb.redirectOutput(output);

pb.redirectError(err);
```

siendo pb el objeto de la clase ProcessBuilder.
Por otro lado, esta aplicación la ejecuté únicamente en modo pseudo distribuido. Para poder hacerlo, tuve que modificar las rutas a los archivos de entrada y salida a HadoopIter. Aunque no tiene en principio mayor complicación, en el apartado de conclusiones explico ciertos problemas que me encontré al hacerlo.

6.5 - Conclusiones

La palabra que me viene a la mente cuando rememoro el trabajo realizado para este capítulo es "problemas". Me he enfrentado a una serie de retos realizando esta parte del trabajo que, aunque a priori no parecían ser excesivamente complejos, me han proporcionado largas horas de trabajo para poder llegar a buen puerto. En principio, como la decisión había sido no construir el generador de tráfico genérico, pensé que la solución iba a ser introducir las aplicaciones ya construidas dentro de un bucle y poco más. Como todo en esta vida, la realidad resultó ser mucho más compleja y, a la par, didáctica y entretenida.

El apartado de introducir el código dentro de un bucle, en función de unos parámetros de entrada no tuvo mayores complicaciones. Sin embargo, tras cada ejecución había que llamar al servidor de históricos para obtener las métricas, y aquí es donde empezaron los problemas.

Ya he comentado en apartados anteriores la necesidad de trabajar con los archivos en formato JSON. Existen numerosas librerías Java para hacerlo, pero me decanté por las javax.json, como ya he dicho, ya que son muy empleadas y porque ya existe la dependencia para Maven en los repositorios oficiales. Tenemos la posibilidad de crear dependencias para las bibliotecas que no existan en dichos repositorios, pero es añadir aún más carga de trabajo.
Como ya he dicho en el apartado correspondiente, no empleé un parser sino que traté los archivos JSON de forma directa, para lo que tuve que estudiar la estructura de los mismos detalladamente para saber cómo y dónde obtener las métricas adecuadas. Para ello, a veces bastó con llamar al servidor de históricos desde el navegador, mientras que en otras ocasiones tuve que escribir una pequeña aplicación que escribiese el archivo leído del servidor en un archivo de texto en el que estudiar la estructura del JSON.

Para acceder a los archivos JSON de los servidores programáticamente tuve que estudiar las REST APIs de MapReduce y Spark. Las tenemos disponibles, respectivamente, en los enlaces:

https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html

http://spark.apache.org/docs/latest/monitoring.html

También tuvo su intríngulis encontrar la dirección correcta del archivo JSON concreto en el servidor de históricos. La documentación oficial de Hadoop en general puede llegar a ser bastante parca y críptica en algunos casos.

Otro problema que ya he comentado fue el de obtener las identidades de los trabajos ejecutados, tanto en MapReduce como en Spark. No abundaré en el mismo, ya que lo he detallado más arriba. Tan sólo

comentaré que el tiempo dedicado mereció la pena, puesto que me permitió a la postre el construir el generador de tráfico genérico, además de profundizar en el tratamiento de archivos JSON.

Por último, la ejecución del generador de tráfico genérico en modo pseudo distribuido también me proporcionó mucha actividad. Para empezar, tuve que modificar el código de HadoopIter para modificar las rutas a los archivos de entrada. En principio no tiene una mayor complicación, aunque hay que tener claro dónde colocar los archivos para que sean visibles al programa.
En general, el trabajo con el HDFS es bastante engorroso desde la consola debido a que los comandos son bastante largos, incluso para realizar las tareas más simples. Por ejemplo, para ver el contenido del directorio raíz del HDFS hay que escribir:

$ bin/hdfs dfs –ls / (Hadoop 1)

$ bin/hadoop dfs –ls / (Hadoop 2)

Cuando tenemos que realizar comprobaciones continuadas del contenido de subdirectorios, o ver el contenido de ciertos archivos, la cosa es de lo más aburrida. No digamos el copiar archivos de los directorios locales al HDFS, para lo que tenemos que escribir:

$ bin/hdfs dfs –copyFromLocal system.txt /input0/system.txt

Nuevamente, en este capítulo tampoco incluyo bibliografía. No he tenido que emplear documentación adicional, aparte de los enlaces a las REST APIs ya comentados y las APIs de Java para usar las clases que me eran desconocidas.

Puede verse que no existe un generador de tráfico para la aplicación SparkIter. Aunque ya lo explicaré más detalladamente en el capítulo siguiente, dedicado a la realización de los experimentos, el resultado decepcionante en la obtención de métricas significativas para los experimentos que tenía en mente en Spark me hizo volver la vista hacia mi cuenta del Mare Nostrum. En esta computadora está disponible la aplicación SPARK4MN, que nos permite un deployment automatizado de una aplicación de Spark en un clúster, a la par que nos proporciona una serie de métricas muy interesantes.
Debido a ello no construí un generador de tráfico para SparkIter, aunque sería bastante sencillo modificar SparkTrafficGen para que llevase a cabo esta tarea.

Capítulo 7

Pruebas experimentales

7.1 - Índice del capítulo 7

7.1 – Índice del capítulo 7

7.2 – Introducción

7.3 – HadoopLog en Hadoop 1 y Hadoop 2

7.4 – HadoopIter en Hadoop 2

7.5 – SparkLog y SparkIter sobre SPARK4MN

7.6 – Conclusiones

7.2 - Introducción.

Llegados a este punto, hay que comenzar a ejecutar las pruebas de rendimiento de las aplicaciones desarrolladas hasta ahora para ver a qué conclusiones podemos llegar acerca de los modelos MapReduce y Spark. Para ello, iremos realizando una serie de ejecuciones variando el tamaño del problema y tomando métricas significativas que nos permitan llegar a conclusiones igualmente significativas.

Las pruebas se realizaron sobre una máquina virtual generada sobre VirtualBox. Se le asignaron 6 procesadores monohilo (la máquina física tiene un i7-3970X con seis núcleos y 12 hilos), 12 GB de memoria física y 100 GB de espacio de disco.

Los experimentos a realizar son:

- En primer lugar llevaré a cabo una serie de ejecuciones de HadoopLog tanto en Hadoop 1 como en Hadoop 2. Como línea base para la comparación emplearé el archivo **access_log_Aug95**, descargado del enlace que se menciona en el apartado 2.5 del presente trabajo. En dicho apartado comentaba que iba a emplear los archivos descargados concatenados varias veces hasta llegar a los 1,5 GB y casi 14 millones de líneas de texto. Aunque para comprobar el correcto funcionamiento de las aplicaciones ha sido más que suficiente, finalmente, y para obtener resultados más representativos, he concatenado el archivo anterior 2, 4, 8, 16, 32 y 64 veces hasta llegar a un archivo de más de 10 GB y más de 100 millones de líneas de registro.
- Otro experimento será efectuar una serie de ejecuciones de la aplicación HadoopIter para un número determinado de iteraciones en cada caso. Mi idea era también variar el número de ecuaciones del sistema. Como ya he comentado en el capítulo 4, es complicado encontrar sistemas de ecuaciones grandes convergentes para el método de Jacobi. Emplearé un sistema de 4 ecuaciones con 4 incógnitas, aunque para un sistema como Hadoop tendría más sentido un número muy elevado de las mismas dentro de un archivo en el HDFS. Emplearé como tamaño de problema el número de iteraciones.
- El tercer juego de pruebas consistirá en repetir las pruebas realizadas sobre HadoopLog en SparkLog. En este caso, las pruebas se realizarán sobre el computador del BSC Mare Nostrum III, empleando la aplicación SPARK4MN. Esta aplicación realiza un “auto deploy” de un clúster de Spark sobre un número de nodos especificado en un script de entrada. El “baseline” es el mismo que en el caso de HadoopLog, pero aquí he podido ir hasta 128 veces este valor implicando un archivo de 21 GB de peso y 200 millones de líneas de registro. Para poder hablar del propio funcionamiento de SPARK4MN, realizo las mismas pruebas solicitando en un caso 1 nodo al sistema y 8 en otro.
- El cuarto y último conjunto de pruebas consiste en la ejecución de SparkIter sobre el mencionado SPARK4MN. El número elegido para las iteraciones ha sido 25, 50, 100, 150, 200, 250, 300, 350 y 1000. Podemos ver que es muy superior al elegido para MapReduce. También efectúo en este caso las mismas pruebas sobre un nodo y sobre 8.

7.3 – HadoopLog en Hadoop 1 y Hadoop 2

Comentaba en el apartado anterior que iba a ejecutar HadoopLog en las versiones 1 y 2 de Hadoop. La motivación de ejecutar las mismas pruebas tanto en Hadoop 1 (versión 1.2.1) como en Hadoop 2 (versión 2.7.1) de esta aplicación para MapReduce ya se ha comentado a lo largo de este trabajo, y no es otra que comprobar si existe diferencia entre el planificador original de Hadoop integrado en MapReduce y el nuevo sistema que se ejecuta sobre YARN. Al contrario de lo que sucede con las aplicaciones desarrolladas para Spark, que podemos ejecutar sobre un clúster real, estas pruebas se ejecutarán en mi caja personal en modo semi distribuido. Este es un hándicap que sesgará el resultado de las pruebas y que no será plenamente extrapolable a un sistema distribuido real. Puede servir, no obstante, de ejercicio metodológico para mostrar una forma de evidenciar la diferencia entre ambos planificadores, si es que existe.

Los resultados para la ejecución en Hadoop 1 son:

	lineas archivo	elapsed time	exec time	cpu time	map_millis	red_millis	physical mem (bytes)	KB read	KB written	Maps
baseline	1569898	17140	16874	12570	12086	9459	863375360	163889	1855	3
x2	3139796	22161	21866	22590	20386	15295	1388728320	327778	1855	5
x4	6279592	29575	29466	42560	38253	22126	2623688704	655560	1855	10
x8	12559184	51227	51109	82240	75175	44454	4999929856	1311123	1855	20
x16	25118368	90513	90391	161370	145191	83681	9708732416	2622250	1855	40
x32	50236736	193201	193100	338370	327489	181999	19183325184	5244505	1855	80
x64	100473472	456508	456231	709450	754801	438419	38106177536	10489013	1855	160

Tabla de resultados de la ejecución de HadoopLog en Hadoop 1.

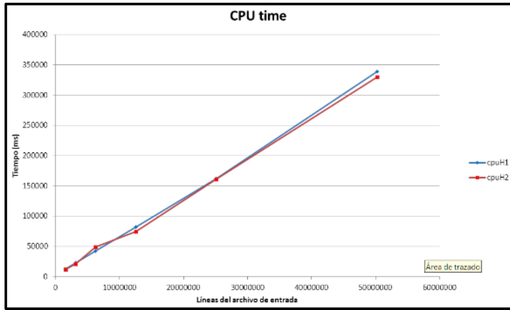
Mientras que los resultados para Hadoop 2 son:

	lineas archivo	elapsed time	exec time	cpu time	map_millis	red_millis	physical mem (bytes)	KB read	KB written	Maps
baseline	1569898	20930	15294	12080	10595	6006	759980032	163885	1855	2
x2	3139796	22247	18318	21160	20079	8002	1105174528	327770	1855	3
x4	6279592	30954	27505	49020	56178	11210	1620660224	655539	1855	5
x8	12559184	41494	37119	74630	98220	21821	3012145152	1311082	1855	10
x16	25118368	85544	81955	160590	292080	53707	5761753088	2622168	1855	20
x32	50236736	185270	180808	330020	567846	147218	11302526976	5244340	1855	40
x64	100473472									80

Tabla de resultados de la ejecución de HadoopLog en Hadoop 2.

Podemos ver que la ejecución de baselinex64 sobre Hadoop 2 no se ha completado. Tras realizar varios intentos, la ejecución se bloqueaba y no llegaba a completarse, por lo que desistí finalmente. Sospecho que no se ejecutaba por limitaciones en la configuración de la máquina virtual. De todos modos, creo que la tendencia que se puede ver en los gráficos es suficientemente representativa como para efectuar una comparación.

En primer lugar, me parece interesante ver el gráfico de tiempo de CPU en ambos modelos:



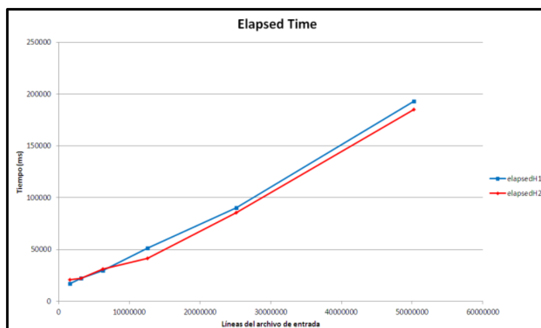
Gráfica del tiempo de CPU de HadoopLog.

La representación tiene líneas de archivo en las abscisas frente a tiempo de CPU (en ms) en las ordenadas.
Aunque quizás no podamos decir que ambos tiempos son calcados, sí que se aproximan enormemente.

Aunque parece despegarse un poco para baselinex32, también lo hacía en x8 y en 16 volvían a encontrarse.

No considero sorpresivo este resultado (ni ninguno de los posteriores). Las modificaciones realizadas en Hadoop 2 se centraban en una mejor planificación de las tareas, a la par que se separaba esta tarea de MapReduce para crear YARN, un planificador genérico que permite ejecutar sobre él otros modelos, como Spark. Que el tiempo de CPU de una misma aplicación sobre ambos modelos sea el mismo no debe sorprendernos.

Veremos a continuación los tiempos totales y de ejecución:



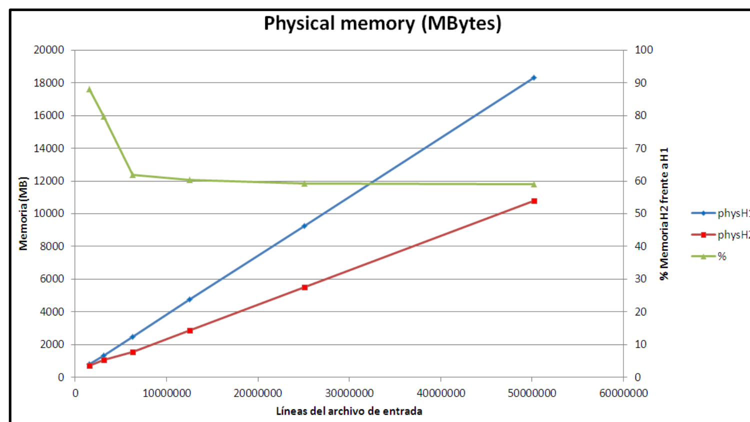
Tiempo total de HadoopLog en función del número de líneas.

Tiempo de ejecución de HadoopLog en función del número de líneas.

Aunque no hay una diferencia espectacular, vemos que el tiempo total y de ejecución tienden a separarse según aumenta el tamaño de problema N. Aquí echo de menos una ejecución en clúster para tener un resultado más consistente. Casi podemos decir que la diferencia de tiempo es constante según los gráficos, aunque si dibujásemos líneas de tendencia veríamos que van divergiendo cada vez más.

La conclusión es que, aunque el tiempo de CPU es el mismo, los tiempos total y de ejecución son menores debido a una mejor planificación. Además, si nos fijamos en el número de “mappers” en ambos entornos, vemos que en Hadoop 2 son prácticamente la mitad que en Hadoop 1. Esto implica el empleo de un menor número de nodos con un mejor aprovechamiento de los recursos de cada nodo. El planificador no se limita a emplear más “mappers” para efectuar la computación distribuida, sino que trata de optimizar el número de los mismos frente a un mejor aprovechamiento de los recursos de cada nodo. Todo apunta a que en un clúster la diferencia sería netamente mayor, puesto que un menor número de nodos puede estar más próximo al master y la comunicación de resultados al mismo puede efectuarse en un tiempo menor.

Para terminar he dejado el resultado más significativo. El uso de memoria física:



Memoria consumida por HadoopLog en función del número de líneas del archivo de entrada.

Un mejor cálculo de los recursos necesarios por parte del planificador, nos permite ajustar mejor el tamaño necesario en cada nodo, por lo que la reserva de memoria física que hace YARN es menor (mucho menor, como puede verse) y el aprovechamiento de los recursos está más optimizado. Podemos lanzar más trabajos en el clúster en Hadoop 2 que en Hadoop 1 gracias a YARN.

He comentado que el fallo de la ejecución de baselinex64 sobre Hadoop 2 se debía a limitaciones en la configuración de la máquina virtual. Si YARN efectúa un mejor uso de los recursos, ¿cómo es esto consistente con mi explicación? Porque la limitación principal de MapReduce es el uso intensivo del sistema de archivos y, aunque en las tablas podemos ver que son muy similares, el menor número de "mappers" ejecutándose de modo más intensivo llega a saturar el sistema.

Un experimento interesante podría ser efectuar las ejecuciones forzando el número de "mappers" en ambos entornos para que fuesen iguales, pero me quedo sin tiempo para profundizar en ciertos experimentos.

Creo que debo profundizar en a qué me refiero cuando hablo de un uso más intensivo de los recursos del nodo. En Hadoop 1 se trabajaba con la noción de "slots", que eran configuraciones estáticas específicas de cada nodo (los nodos no tienen por qué ser iguales), que determinaban el número máximo de procesos map y/o reduce que se podían ejecutar en cada nodo. Esto puede llevar a que el nodo esté infra utilizado ya que, debido a su naturaleza estática, puede que no "quepa" un nuevo slot en el nodo pero existan recursos libres. Por el contrario, la única limitación de YARN es la memoria máxima disponible en el nodo. YARN trabaja con el concepto de "containers", que es mucho más flexible en cuanto a gestión de recursos que los "slots" de Hadoop 1.
Además, los "slots" tenían un límite máximo por lo que, típicamente, se procuraba que el número de "slots" multiplicado por la memoria máxima configurada para cada uno de ellos fuese inferior a la memoria máxima disponible en cada nodo, lo que limita la capacidad de ejecutar trabajos con un uso de memoria intensivo.
Dicho esto, puesto que cada "mapper" se ubica en un slot, la rigidez de los mismos nos obliga a crear un mayor número de "mappers" para efectuar la misma tarea que los creados en los contenedores de YARN, más flexibles y que pueden ser ubicados en nodos con recursos sobrantes más limitados. En resumen, menos "mappers" y menos memoria física total necesaria.

Para terminar con este tema, el eje secundario del gráfico representa el porcentaje de memoria utilizada en Hadoop 2 respecto de la utilizada en Hadoop 1. Vemos en la tabla de datos que con tamaños pequeños de N, el número de "mappers" es muy similar, y la diferencia de memoria es relativamente pequeña. Sin embargo, a partir de un cierto tamaño, el número de "mappers" empleado por Hadoop 2 es la mitad de los empleados por Hadoop 1, y la memoria empleada por H2 se estabiliza en torno al 60% de la empleada por H1.

7.4 – HadoopIter en Hadoop 2

Aunque en la descripción del experimento lamento no haber encontrado sistemas de ecuaciones más grandes, el objetivo del mismo es comprobar el aumento de accesos al HDFS, que es el principal motivo de la ineficiencia de MapReduce en la ejecución de trabajos iterativos. Aunque no realizo una comparación directa entre MapReduce y Spark, sí podremos ver la enorme diferencia en este apartado, cuyos resultados son idénticos a los que obtendríamos trabajando con cualquier otro sistema.

Un detalle que no habrá pasado desapercibido al lector es que este experimento se realiza únicamente sobre Hadoop 2. En el apartado anterior ya hemos visto las diferencias entre ambos entornos y no es esperable variaciones en este sentido, aunque el motivo no es éste. Mi intención era llevar a cabo ejecuciones en ambos entornos que me permitiesen una comprensión más profunda de las diferencias entre ellos pero, una vez más, un problema técnico se interpuso en mi camino.

La aplicación HadoopIter la desarrollé sobre Hadoop 2. Podemos ver en el código que cada reducer efectúa una escritura sobre el archivo de estimas "guess", para lo que abrimos dicho archivo en modo "append". En el código:

FSDataOutputStream guessWrite = fs.append(guessFile);

El modo "append" está soportado (de modo fiable) en Hadoop 2, pero en Hadoop 1 sufrió una serie de altibajos. Dicho modo no era muy estable en Hadoop 1, y dejó de soportarse en las versiones 1.x porque producía resultados impredecibles en HDFS, aunque sí se soportaba en las versiones 1.0.x. En teoría, se podía activar dicha funcionalidad en Hadoop 1.2.1 modificando la variable dfs.support.broken.append. Tras varios intentos, seguía obteniendo el mismo error de "append" no soportado en las ejecuciones, aunque hice que el programa me mostrase el valor de la variable antes de llegar a la apertura del archivo en este modo. Como no me rindo fácilmente, probé con versiones de Hadoop 1.0.x en las que sí estaba soportada esta funcionalidad. En ellas me encontré con que otras funcionalidades empleadas en el código no estaban soportadas por ser modificaciones llevadas a cabo en versiones posteriores. Esto es lo que me hizo desistir finalmente de ejecutar HadoopIter sobre Hadoop 1.

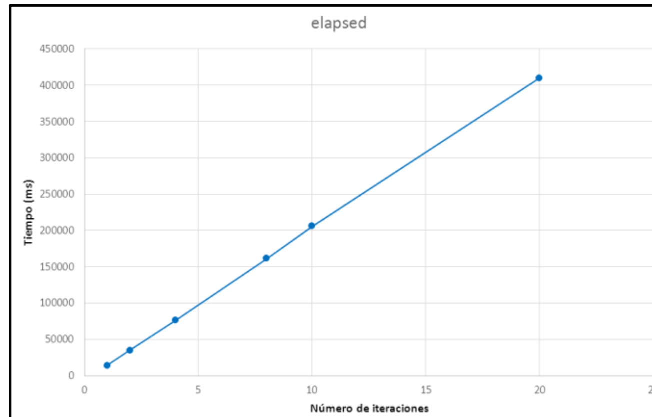
Cuando construí esta aplicación no realicé muchas ejecuciones de prueba, pero sí cuando construí SparkIter. Debido a ello, mi idea inicial del número de iteraciones rondaba el empezar las pruebas con 25 y terminar con 400 (en SparkIter 1000 iteraciones se ejecutan en 18 segundos en modo local). En cuanto inicié las pruebas, me di cuenta que 25 iteraciones de HadoopIter en modo semi distribuido llevaban un tiempo bastante importante, por lo que decidí que las pruebas serían con un número de iteraciones de 1, 2, 4, 8, 10 y 20 respectivamente.

La tabla de los resultados obtenidos en estas pruebas es la siguiente:

iteraciones	elapsed time	Max physical mem (bytes)	bytes read HDFS	bytes written HDFS
1	14021	433684480	327	196
2	34625	437555200	739	392
4	76098	438321152	1563	784
8	161176	436408320	3296	1568
10	205975	438206464	4120	1960
20	410090	437325824	8260	3920

Tabla de resultados de la ejecución de HadoopIter

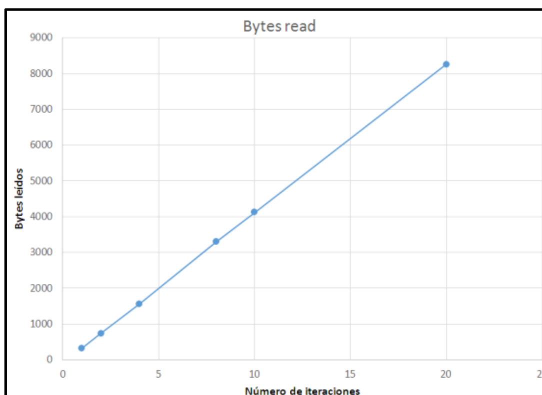
Este paquete de pruebas no ha producido resultados sorprendentes, lo cual no tiene por qué ser necesariamente malo, por lo que trataré los resultados de forma liviana. Empezaré por el tiempo total empleado por la aplicación:



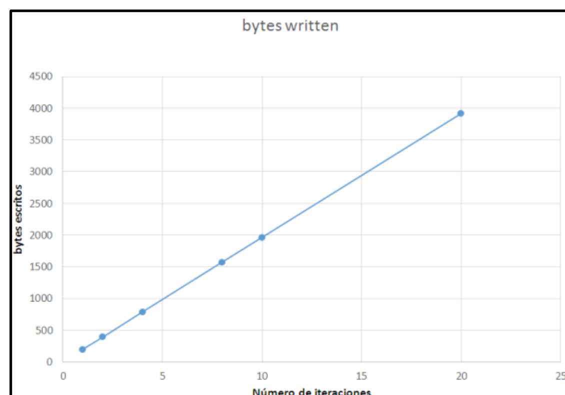
Tiempo total de HadoopIter en función del número de iteraciones.

Debemos recordar que hemos implementado el trabajo iterativo en MapReduce como una serie de trabajos MapReduce, donde la salida de uno sirve de entrada al siguiente. Todos ellos son trabajos idénticos que realizan las mismas operaciones sobre unos datos iguales, o del mismo tipo, de crear los mismos contextos para la ejecución de los trabajos y de realizar lecturas y escrituras equivalentes en HDFS. No es de extrañar, por lo tanto, La relación lineal entre el tiempo de ejecución y el número de iteraciones.

Puesto que hemos dicho que los diferentes trabajos realizan lecturas y escrituras equivalentes en el sistema de archivos, sería de esperar que la relación entre los bytes escritos y leídos también fuese lineal respecto del número de iteraciones. En los gráficos que siguen podemos comprobar que es así.



Número de bytes leídos por HadoopIter del HDFS por número de iteraciones



Número de bytes escritos por HadoopIter en el HDFS por número de iteraciones

No realizaré gráficos de la memoria máxima empleada. Como ya he comentado, cada ejecución consiste en correr N veces la misma aplicación MapReduce, con la única diferencia de que la solución final es más refinada al terminarse cada uno de los trabajos que componen la ejecución. Como es lógico pensar, la memoria máxima, sin ser la misma, es muy parecida en todos los casos y no merece más comentarios.

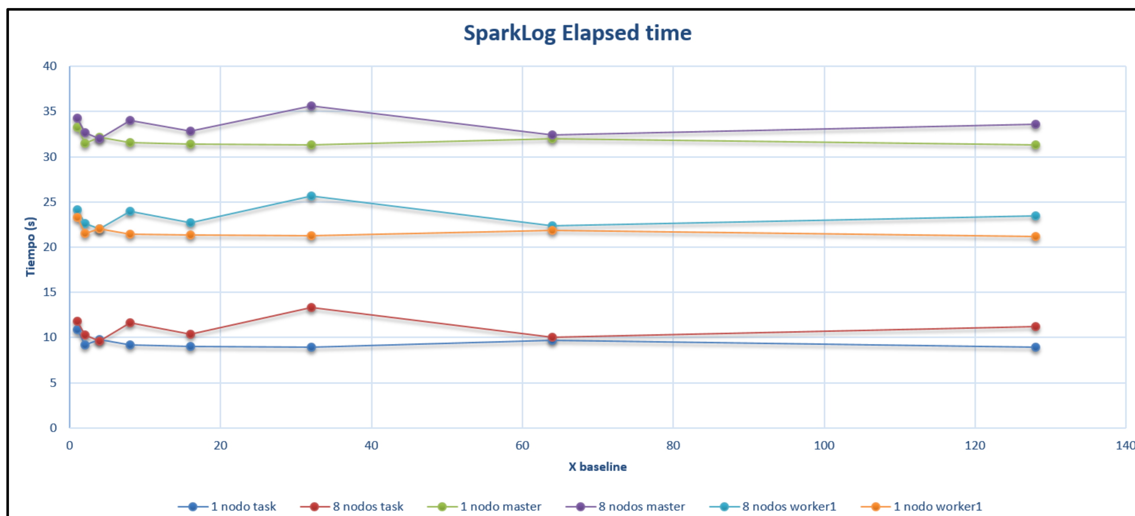
7.5 – SparkLog y SparkIter sobre SPARK4MN

El uso del recurso SPARK4MN ha representado una muy interesante oportunidad no sólo de trabajar en un clúster real, sino la de emplear un entorno de súper computación que me ha proporcionado unos resultados interesantes, aunque necesitados de una interpretación diferente.

Puede deducirse del propio título de este apartado que he modificado la forma de explicar los resultados, al incluir ambas aplicaciones en un mismo apartado. Tenía pensado seguir la misma metodología que con las aplicaciones de MapReduce, pero a tenor de los resultados me parece más lógico e interesante hacerlo así, puesto que me da cierta perspectiva para hablar del propio funcionamiento de la aplicación SPARK4MN.

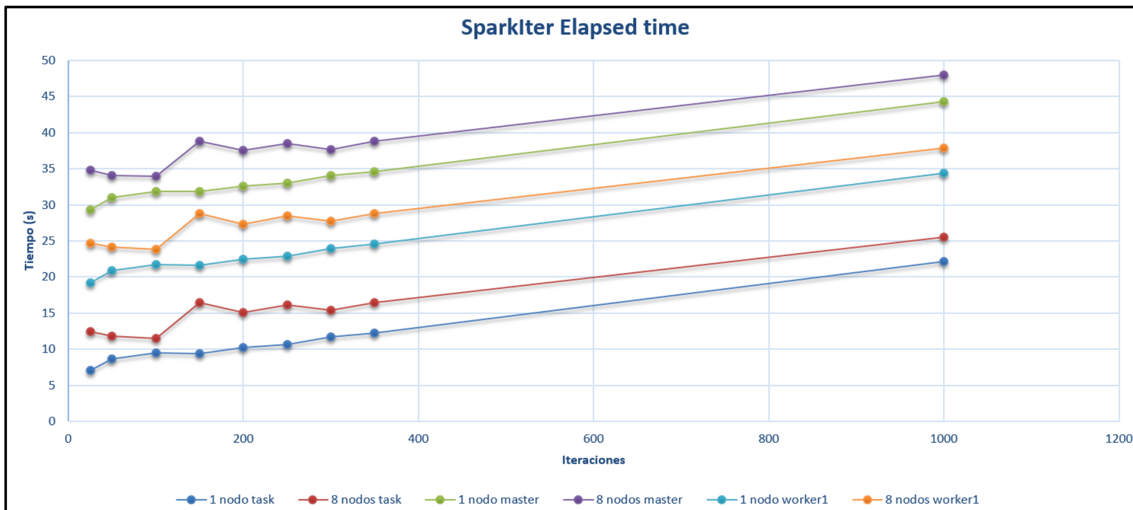
En este caso presentaré en primer lugar los tiempos totales de ejecución de las aplicaciones. Podemos ver a continuación dos gráficas. En ambas hay mucha información, puesto que represento el tiempo total empleado por la aplicación, entendido como el tiempo empleado por el comando **spark submit…**, además del tiempo empleado por el master de la aplicación (incluido su “deployment”), así como el tiempo empleado por el worker1 (incluido también su “deployment”). En el caso de trabajo con 8 nodos, tan sólo represento el tiempo empleado por el worker 1 porque, aunque existen 8 ya que se han solicitado 8 nodos, el tiempo empleado por todos ellos es prácticamente idéntico.

El gráfico correspondiente a SparkLog es el siguiente:



Diferentes tiempos totales de SparkLog según el número de nodos y el tamaño del archivo de entrada.

Mientras que la gráfica correspondiente a SparkIter es:



Diferentes tiempos totales de SparkIter según el número de nodos y el tamaño del archivo de entrada.

Podemos ver que, tanto en el caso de SparkLog como en el de SparkIter tenemos, para 8 nodos, un punto en el que el tiempo total es mayor que el que podría esperarse según vemos la tendencia. Cuando realicé las pruebas con SparkLog pensé que podría ser un "outlier", pero al ver que el resultado se repetía con SaprkIter cambié de idea. Seguramente lo que suceda es que en esos puntos de ambas aplicaciones la gestión de los 8 nodos (el "overhead" por paralelismo) sea más costosa por la propia naturaleza de cada aplicación, lo que hace que el tiempo total necesario sea mayor.

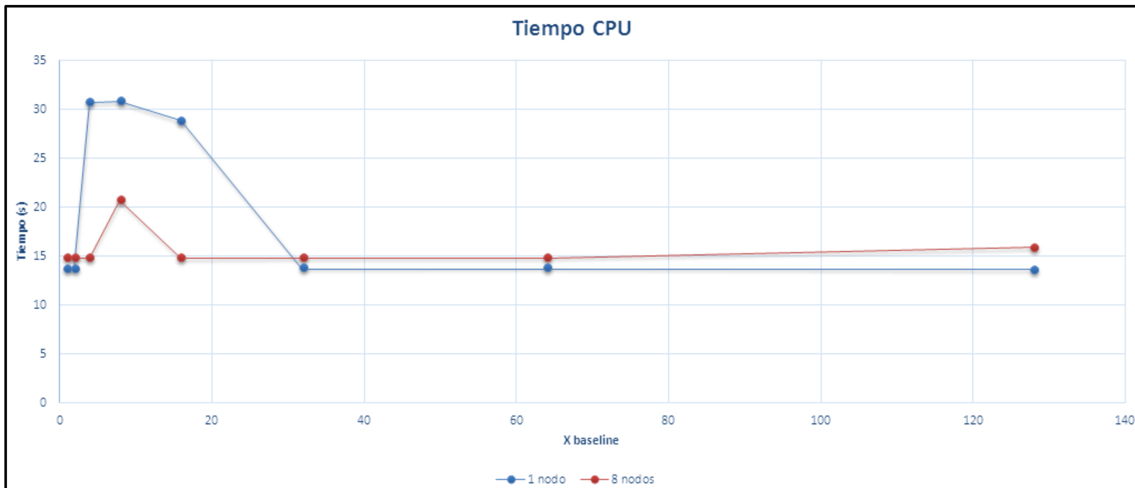
Además podemos ver que en todos los casos el tiempo total es superior en el caso de los 8 nodos. En Spark ambas aplicaciones se ejecutan muy deprisa, por lo que podría pensar que el "overhead" por paralelismo sea mayor que el propio tiempo de ejecución por ser una aplicación que precisa de poco tiempo para ejecutarse. A pesar de que la tendencia parece mantenerse para mayores valores de N, pienso que es una sensación un poco artificial al pasar en la gráfica de 350 iteraciones a 1000, y para valores realmente grandes de N esta tendencia debería desaparecer.
Asimismo, el sistema de ecuaciones es muy pequeño y me gustaría poder haber trabajado con sistemas realmente grandes. Seguramente, para sistemas de ecuaciones mayores el "overhead" por paralelismo se habría compensado sobradamente.

En el caso de SparkIter vemos que el tiempo total va aumentando con la complejidad del problema. Aunque en el caso del "master" y del "worker1" están incluidos los tiempos de "deployment", asumo que estos tiempos son siempre los mismos, y la tendencia al alza la marca la complejidad del problema. El que obtenemos es un resultado esperable, ya que a mayor N creamos una mayor necesidad de procesamiento y el tiempo total aumenta. Veremos en el tiempo de CPU que la tendencia es la misma.

Sorprendente, sin embargo, el resultado del tiempo total para SparkLog. Mayor tiempo en las ejecuciones en 8 nodos, a lo que podríamos dar la misma explicación que en el caso anterior, pero tanto en las ejecución en 1 nodo como en la de 8 obtenemos una gráfica de tiempo total bastante plana. Resultado que, sin poder decir que se calca, también aparece en el tiempo total de CPU. Si volvemos la vista hacia los datos de los tiempos totales, de ejecución y de CPU de las aplicaciones para MapReduce, vemos que los tiempos de CPU eran significativos respecto de los tiempos totales, por lo que debemos preguntarnos el porqué de este resultado. La explicación no es otra que los tiempos de CPU que nos ofrece MareNostrum para SPARK4MN son los tiempos totales empleados, precisamente, por SPARK4MN. En este caso, el tiempo de ejecución de la aplicación SparkLog es mucho menos significativo respecto del tiempo necesario para arrancar SPARK4MN y de realizar los "deployments", por lo que parece que un aumento de N no supone un incremento del tiempo total, pero es de esperar que para valores lo suficientemente elevados de N veríamos la misma tendencia que en el caso de SparkIter. SparkIter es una aplicación que realiza un uso más intensivo de la CPU, por lo que su influencia es mayor respecto del tiempo total de SPARK4MN.

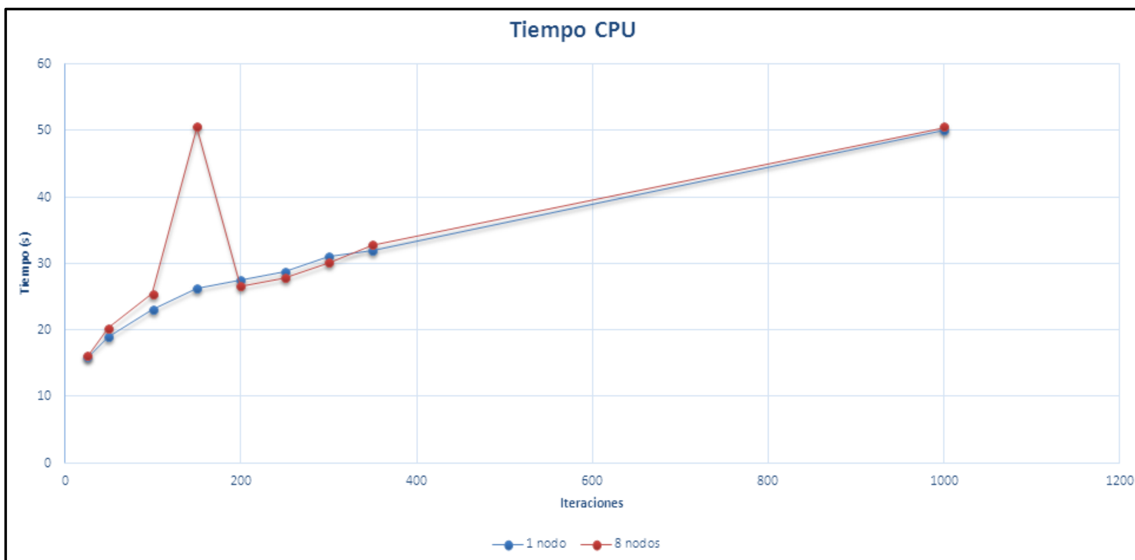
Puesto que hemos estado hablando de los tiempos de CPU y de lo que representan, vemos las gráficas correspondientes.

Para SparkLog:



Tiempo de CPU de SparkLog según el número de nodos y el tamaño del archivo de entrada.

Para SparkIter:



Diferentes tiempos totales de SparkIter según el número de nodos y el número de iteraciones.

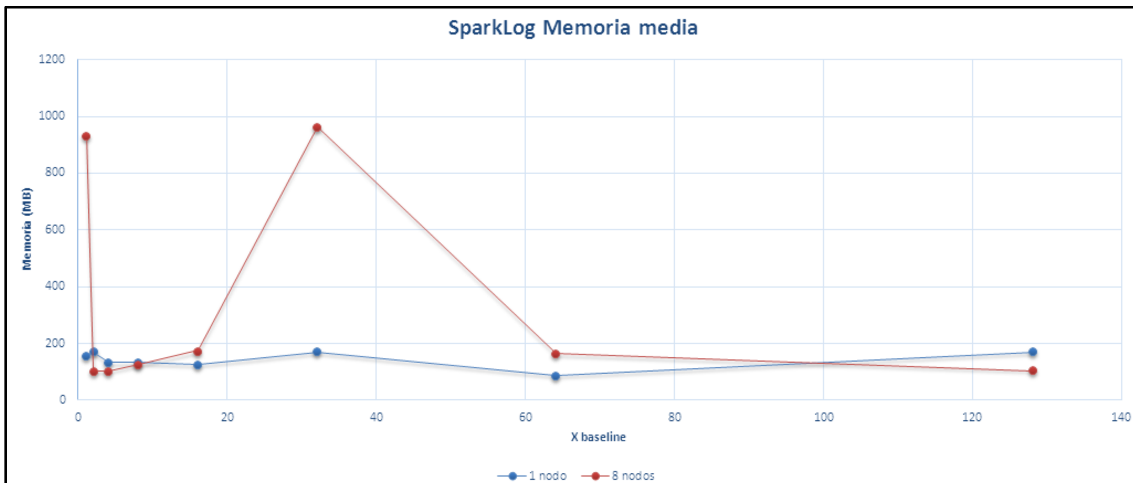
Nuevamente, y como hemos comentado anteriormente, los resultados esperables para SparkIter (con presunto "outlier" incluido) y los resultados extraños en el caso de SparkLog. Seguimos viendo en ambos casos una mayor necesidad de CPU en un momento determinado. Es más, podemos ver que en el caso de SparkLog este comportamiento se produce para un rango de valores de N. La explicación que veo es la misma que en el caso anterior. Por la naturaleza de la aplicación y de los valores de entrada, existe un punto en el que el "overhead" por paralelismo se dispara, volviendo a valores más normales la aumentar N. Hubiera sido interesante repetir los experimentos con una mayor variedad de tamaños de clúster para ver si este fenómeno se repetía y las causas. Podría ser incluso material para un TFG per se.

Vemos más claramente la mayor influencia de una aplicación con un uso intensivo de CPU sobre el tiempo total de CPU de SPARK4MN, mientras que la influencia de SparkLog en el tiempo de CPU es inexistente para los valores con los que trabajamos. Me parece significativo que mientras que los valores

de CPU para el caso de SparkIter van confluyendo, en el caso de SparkLog siempre están por encima los de la ejecución en 8 nodos. La intensidad de uso de la CPU en el caso de SparkIter va compensando el "overhead" por paralelismo, mientras que SparkLog no llega a hacerlo, y el uso de un clúster mayor no se ve compensado (para valores "ineresantes" de N, que es el objeto de Hadoop).

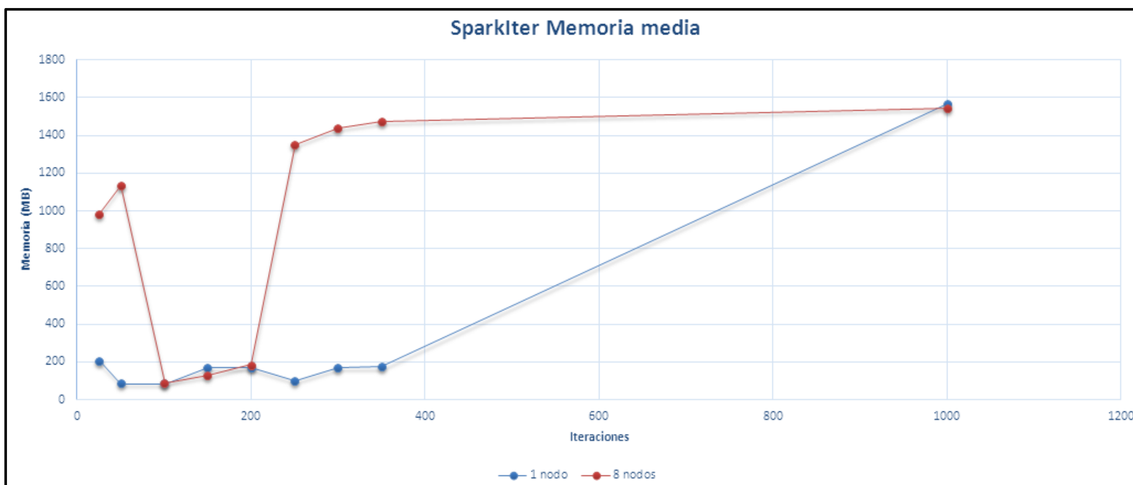
Por último y para terminar con este apartado, veremos las necesidades de memoria de las aplicaciones. Las métricas que me proporciona el Mare Nostrum son la memoria máxima y la memoria media necesitadas por la aplicación, entendida también como el conjunto de la aplicación SPARK4MN. Debido a que se entiende como las necesidades de la aplicación de "deployment", he creído más representativo el atender tan sólo a la memoria media. Los gráficos son bastante similares, aunque en el caso de la memoria máxima existían ciertos picos que atribuyo al sistema en sí y no a las aplicaciones que estoy estudiando. Veamos las gráficas.

Para SparkLog:



Memoria media consumida por SparkLog según el número de nodos y en función del tamaño del archivo de entrada.

Y SparkIter:



Memoria media consumida por SparkIter según el número de nodos y en función del número de iteraciones.

Lo primero que me llama la atención es que mientras que en SparkLog las necesidades de memoria se mantienen bajas, a excepción de los dos picos, las necesidades de SparkIter crecen de forma significativa para un determinado valor de N, tanto para el caso de 8 nodos como para el caso de 1 nodo. En SparkLog el pico de la primera ejecución sí lo considero un "outlier", ya que no hay motivos para esa desproporcionada necesidad de memoria para esa ejecución, mientras que el segundo pico lo achaco

a esa zona de baja eficiencia a la que me he referido anteriormente, y que muestra que para esa aplicación y ese número de nodos, la configuración del sistema no es buena. Cuando lanzamos SPARK4MN, se crean el número de nodos solicitado más otro nodo que ejecutará el master. Aunque tengamos 8 nodos reservados y con los "workers" listos, Spark no va a crear más contenedores de los que considere necesarios. Por lo tanto, para el caso de SparkLog por mucho que aumente N Spark, por la forma de ejecución a modo de grafo acíclico dirigido que tiene (DAG), no considera necesario crear más contenedores para realizar el trabajo. Sin embargo, SparkIter sí que va precisando de más memoria al poder el sistema paralelizar mejor el trabajo y encargar la creación de más contenedores que lo lleven a cabo. Vemos que para el caso de 8 nodos el aumento de la memoria necesaria parece que va a saltos (sería necesario realizar más pruebas con mayores valores de N). Lo interpreto porque hasta cierto valor de N lo contenedores creados son suficientes, mientras que a partir de ahí es necesario crear más, lo que hace que aparezca un "salto" en la gráfica.

Para que esta explicación sea más verosímil debo hablar de ciertos problemas que tuve con la ejecución de mis aplicaciones en SPARK4MN. Tras realizar las pruebas locales de las que hablo en los apartados correspondientes, llegó el momento de migrar mis aplicaciones a SPARK4MN. Las primeras ejecuciones se saldaron con la aplicación SparkIter (la primera que traté de migrar) entrando en bucle y no terminando hasta llegar al límite de ejecución solicitado al sistema. Tras mucho revisar la aplicación, decidí solicitar ayuda al centro de soporte de Mare Nostrum, quienes me indicaron que la memoria que solicitaba para los contenedores de Spark en mi script de configuración de SPARK4MN era muy pequeña, recomendándome que pidiese el máximo admitido. Así lo hice, ejecutándose mis aplicaciones correctamente a partir de ese momento. Debido a la gran cantidad de memoria que se reserva para cada contenedor, cuando se crea un contenedor nuevo para absorber esa mayor carga que supone un aumento de N, vemos en el gráfico que memoria necesaria por el sistema aumenta de golpe.
La gráfica, aunque engañosa por el salto de 350 a 100 iteraciones, me hace pensar que en algún punto entre estos dos valores sucede lo mismo para un nodo y que, con sucesivos aumentos de N, iríamos viendo escalones en la gráfica. Este valor también es configurable en cualquier versión de Spark y hubiese sido interesante realizar pruebas con diferentes configuraciones de memoria para ver cómo se producían los incrementos de memoria necesaria por la aplicación.

7.6 – Conclusiones

No creo que tenga que decirse que este ha sido el capítulo más grato no ya de escribir, sino de ejecutar. Aquí he visto el fruto de meses de trabajo y, aunque tenía en mente una serie de resultados e interpretaciones que he tenido que variar para amoldarme a las métricas disponibles, los resultados me han proporcionado un gran enriquecimiento a nivel didáctico y personal.

Los resultados de las aplicaciones para MapReduce me han permitido una mejor comprensión de las arquitecturas de Hadoop 1 y Hadoop 2, así como de las mejoras llevadas a cabo en este último respecto de su predecesor.

El trabajo con SPARK4MN me ha permitido entrar en contacto con un sistema de computación distribuida y me ha exigido tener en cuenta las particularidades propias de estos sistemas a la hora de interpretar los resultados, así como comprender las necesidades de dichos sistemas a la hora de elaborar scripts de configuración y lanzamiento de las aplicaciones en los mismos.

Capítulo 8

Conclusiones sobre el TFG

“MapReduce and YARN are not for the faint of heart…”

Así terminaba el apartado dedicado a la motivación de mi trabajo de fin de grado. Apenas si he arañado la superficie de MapReduce, no digamos ya la de Spark con una API mucho más rica y compleja y sí, puedo afirmar que MapReduce y YARN no son para los débiles de espíritu.

No quiero que este apartado sea una repetición de las conclusiones que he ido obteniendo en los diferentes capítulos. Esas conclusiones ya están ahí y no quisiera repetirlas. Me gustaría emplear este último capítulo para decir cuáles han sido mis impresiones generales a lo largo de los seis meses que me ha llevado la realización de este trabajo. También creo positivo el dividir estas impresiones en dos partes; por un lado las impresiones que podríamos llamar didácticas o académicas, mientras que por otro no menos importante, las impresiones que he tenido a nivel personal.

A nivel académico la experiencia ha sido muy positiva, ya que me ha permitido conocer una serie de tecnologías desconocidas por mí hasta ahora, a la par que adentrarme en el mundo de la computación distribuida.

Hasta ahora, aparte de la asignatura de Sistemas Distribuidos y de las pruebas realizadas en la asignatura de Arquitectura Avanzada de Computadoras con la máquina de la UOC, no había tenido contacto con estos sistemas, por lo que se puede decir que partía de cero en este aspecto. No me había planteado la existencia de modelos de programación diferentes y pensaba que todo era un poco “ad hoc”. Es por ello que el recorrido a través de MapReduce y Spark ha supuesto un grato descubrimiento. Ha sido interesante ver las diferentes arquitecturas de Hadoop, su evolución en el tiempo, los diferentes modos de planificación (que no he tratado en este trabajo) y las radicales diferencias entre los modelos de programación de MapReduce y Spark, así como las diferencias en sus conceptos y aplicaciones.
Las opciones de configuración tanto de MapReduce como de Spark son muy grandes, con un gran número de variables disponibles que nos permiten optimizar la ejecución de nuestras aplciaciones. Apenas si me he atrevido con algunas y hubiera sido interesante entrar con más profundidad en este tema pero, como comento en el capítulo de las pruebas experimentales, aquí hay material para un TFG en sí mismo.

La recopilación de la información fue todo un reto en sí mismo. Hay una gran cantidad de documentación disponible en internet, pero la documentación oficial es parca y oscura en algunos apartados. La documentación en castellano es escasa, y casi limitada únicamente a aspectos introductorios. La bibliografía en inglés es abundante pero si no se compra, que no se puede comprar todo, es de difícil acceso.
Una vez recopilada, hubo que ir saltando de una obra a otra para ir aclarando conceptos que en una no quedaban claros, o para completar o acceder a otros. Pero esto no es diferente de cualquier otro trabajo de estas características.

La construcción de las aplicaciones me resultó ardua. Se puede ver que la extensión de las mismas no es muy grande, pero la novedad de los modelos de programación me hizo atascarme en algunos momentos. La API de MapReduce no es compleja, pero requiere cierto proceso de adaptación. Spark tiene una API más compleja y rica, pero la principal dificultad que me encontré fue la necesidad de trabajar con la interfaz Function en sus diferentes modalidades debido a la elección de la versión 7 de Java. En último término, esa elección dio fruto al permitirme ejecutar las aplicaciones sin apenas modificaciones en SPARK4MN, aunque trabajar con las expresiones lambda no hubiera sido fácil. Hubiera sido interesante trabajar con las expresiones lambda tanto a nivel didáctico, como por el hecho de que existe muchos más ejemplos de código en Scala que en Java, ya que es el lenguaje de desarrollo de Hadoop y en Scala se emplean dichas expresiones. Existen pocos ejemplos en Java, y menos con las implementaciones de Function, ya que se emplea mayoritariamente Java 8 debido a la potencia de las expresiones lambda y su similitud a Scala.

Respecto a las aplicaciones debo decir también que son mejorables, como todo en esta vida. Con más tiempo hubiera implementado un control de excepciones mucho más riguroso, hubiera añadido un

archivo de configuración en XML al generador de tráfico genérico para que pudiera tratar cualquier tipo de JAR y muchas otras cosas que se quedan en el tintero. La idea de crear los generadores de tráfico apareció con el proyecto ya en marcha y tuve que apretar los tiempos de construcción de las aplicaciones, lo que ha redundado en la calidad de las mismas. Sin embargo, en mi opinión, ha merecido la pena el perder esa calidad para explorar esa vía, que me ha aportado un mayor conocimiento de los sistemas estudiados, así como de otros conceptos que no hubiera explorado de no haber construido los generadores de tráfico.

A la hora de trabajar con Spark descubrí que, a pesar de que Hadoop viene con un compilador para crear los archivos JAR, Spark no. Consultando la documentación vi que se pueden emplear diferentes herramientas, entre las que la más usada es Maven. Aunque tuve la tentación de generar los JAR con la aplicación de Hadoop y ver si funcionaban (deberían hacerlo), decidí adentrarme en Maven. Me resultó interesante ver la forma de tratar las bibliotecas necesarias como dependencias y su forma de almacenarlas en un repositorio local. Una vez que uno se acostumbra resulta muy cómodo.

Dentro de los temas que no hubiera tocado de no haber construido los generadores de tráfico está el formato de archivos JSON. Era un formato desconocido para mí que empezó a emplearse con Javascript y que ha trascendido a un número mayor de aplicaciones. En el caso de MapReduce, los archivos que obtenemos del servidor de históricos están tanto en formato XML como en JSON, pero Spark sólo los proporciona en formato JSON, lo que me hizo inclinarme por él. Los conceptos en los que se basa el formato son muy simples, pero permiten construir archivos de una gran complejidad. Existen "parsers" para trabajar con el mismo, pero llegado este punto empezaba a mirar ya en el calendario la fecha de entrega del TFG y decidí emplear javax.json que, aunque no tiene "parser" sí existía como dependencia para Maven y me permitió asimilar la biblioteca más rápidamente.

La elección de Java como lenguaje de programación se basó en mi conocimiento previo del mismo, frente a Scala o Python. Esta decisión me ha permitido conocer en profundidad mis grandes carencias en este lenguaje, así como paliar algunas de ellas. Hasta ahora mi contacto había sido con Java 6, por lo que la necesidad de trabajar en Java 7 y emplear implementaciones de programación funcional me ha resultado costoso, a la par que gratificante. También me ha gustado el breve contacto con las expresiones lambda de Java 8 (y Scala).
Si tuviera que enfrentarme a esta decisión de nuevo, volvería a elegir Java para poder emplear SAPRK4MN, pero en caso de poder elegir libremente me pasaría a Scala por su sintaxis simple y potente.

Para terminar estos comentarios a nivel didáctico quiero decir que me ha gustado el tratar el trabajo como un proyecto. Cuando me planteé la realización del TFG y vi las condiciones necesarias para matricularse en el mismo me pregunté por qué era necesario el haber cursado la asignatura de Gestión de Proyectos. Ahora lo entiendo, pues he tenido que realizar una programación de las tareas a llevar a cabo, he tenido que modificar dicha programación, me he tenido que replantear soluciones e incluso objetivos al encontrarme con problemas técnicos, reescribir porciones de código, parar el trabajo y dedicarme a la formación en un tema que me iba a resultar necesario… En definitiva, un proyecto en miniatura que me ha permitido interiorizar dicha asignatura.

A nivel personal la experiencia ha sido, a partes iguales, gratificante, frustrante y extenuante. Afortunadamente, la parte con al que me quedo es la gratificante.

Digo frustrante porque, como he comentado antes, he tenido que modificar ciertas ideas que tenía inicialmente y que me hubieran gustado llevar a cabo como, por ejemplo, una comparación directa MapReduce – Spark para ver las necesidades de memoria y accesos al HDFS de ambos al ejecutar las mismas aplicaciones. Además, según iba avanzando en el proyecto, especialmente en la realización de las pruebas experimentales, surgían cuestiones interesantes en las que me hubiera gustado profundizar.

Extenuante porque el trabajo que me ha exigido el TFG ha sido muy importante. Creo que puedo decir que ha sido el semestre que más dedicación me ha requerido desde que estoy en la UOC. Pero no creo que haya sido diferente en absoluto a lo que le ha ocurrido a cualquiera de mis compañeros en este sentido.

Sin embargo, a pesar de las frustraciones y la extenuación, el saldo final del TFG a nivel personal no podía ser más positivo. Me ha permitido trabajar en un tema que me apasiona programándome yo las actividades, eligiendo mis tiempos y decidiendo qué hacer o qué no. Me ha permitido profundizar en apartados que conocía o creía conocer y me ha llenado de interrogantes y me ha dejado una serie de actividades en las que trabajar a partir de ahora por pura diversión, sin el "deadline" del TFG pendiendo sobre mi cabeza cuan espada de Damocles.

Francisco Javier Alvarez Goikoetxea
Bilbao – Enero de 2016

Anexo A

Instalación del entorno de trabajo:

- Hadoop 1.2.1
- Hadoop 2.7.1
- Spark 1.4.1

A.1 - Índice del Anexo A

A.1 – Índice del Anexo A

A.2 – Introducción

A.3 – Instalación del JDK-7u79

A.4 – Instalación de Hadoop 1.2.1

A.5 – Instalación de Hadoop 2.7.1

A.6 – Instalación de Spark 1.4.1 con Hadoop 2.6.0

- **A.6.1 – Justificación de la elección**
- **A.6.2 – Proceso de instalación y prueba**

A.7 – Conclusiones

A.8 - Bibliografía

A.2 - Introducción

Para poder llevar a cabo este trabajo, como sucedería con cualquier otro que nos propusiéramos, debemos hacer uso de una serie de herramientas que deberemos instalar previamente en nuestro sistema. Como es evidente, deberemos instalar las herramientas con las que vamos a desarrollar nuestro estudio. Sin embargo, dichas herramientas pueden precisar la instalación previa de una serie de componentes que les permitan llevar a cabo su trabajo.

En primer lugar debemos instalar un JDK, puesto que tanto Hadoop como Spark crean una serie de máquinas virtuales de Java para ejecutar nuestras aplicaciones. Tanto la máquina de la UOC como el Mare Nostrum III tienen instalada la versión 7 de Java, por lo que obviaré la versión 8, más reciente, e instalaré en mi máquina local la versión 7 en su actualización más reciente, que es la 79.

La versión de Hadoop 1 que emplearé será la versión estable más actual, que es la 1.2.1. El criterio de selección para Hadoop 2 es el mismo, y la versión más actual a la hora de comenzar este trabajo es la 2.7.1.
Existen varias distribuciones de Hadoop, tanto comerciales como comunitarias. Para llevar a cabo este trabajo he decidido descargar los binarios de las versiones seleccionadas desde la página de Apache para Linux x86_64, lo que me garantiza que los elementos descargados están actualizados, su gratuidad y un soporte comunitario de gran calidad.

En el caso de Spark, la versión con la que voy a trabajar es la 1.4.1. Aunque ya se ha liberado la versión 1.5.0 a la hora de escribir estas líneas, cuando empecé a trabajar en este proyecto se dieron ciertas circunstancias, que explicaré más abajo, que me indujeron a emplear esta versión.

Aunque para el desarrollo de las aplicaciones de Spark he seleccionado Maven como herramienta, no incluyo la instalación de Maven en este capítulo dedicado al entorno de trabajo del presente proyecto. La razón es que Maven es una de las muchas herramientas que se pueden emplear para desarrollar aplicaciones en Spark e, incluso, el uso de dichas herramientas es opcional.

A.3 - Instalación del JDK-7u79

Para obtener el JDK no tenemos más que ir a la página de Oracle y descargar la versión que deseemos, previa aceptación de las condiciones de uso. En este caso, la página es la siguiente:

http://www.oracle.com/technetwork/es/java/javase/downloads/jdk7-downloads-1880260.html

En nuestro caso, seleccionaremos la versión Linux x64 comprimida en tar.gz. Una vez descargado el archivo comprimido, lo copiaremos al directorio que deseemos y lo descomprimiremos ahí. Si lo hago así es porque en mi sistema ya tengo instalado la versión 8 de Java y, en lugar de desinstalarla, cuando abro una consola para trabajar con Hadoop o Spark ejecuto un script que carga las variables de entorno adecuadas para que usen esta versión de Java.

El script, que he denominado environment.sh, contiene las siguientes líneas:

```
#!/bin/bash
# Environment variables for Hadoop
#
# Francisco J. Alvarez Goikoetxea
#
# TFG - UOC
#
# Execute using: source ./environment.sh

export JAVA_HOME=/home/goiko/programas/jdk1.7.0_79
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

El script tan solo define las variables JAVA_HOME, HADOOP_CLASSPATH y el PATH. Como se indica en los comentarios, debemos ejecutarlo a través de source, es decir:

```
$ source ./environment.sh
```

De este modo nos aseguramos que al llamar a Hadoop, éste encuentre todos los elementos necesarios para su ejecución.

A.4 - Instalación de Hadoop 1.2.1

Uno de los objetivos de este trabajo es comparar el comportamiento de las aplicaciones de MapReduce en Hadoop 1 y Hadoop 2, ya que en la versión 2 MapReduce se ejecuta sobre YARN y el rendimiento de dichas aplicaciones es superior. Por lo tanto, aunque Hadoop se encuentra en su versión 2.7.1, debemos tener instalada una copia de la versión 1.

Ya he comentado que para la realización de este trabajo voy a emplear tres recursos, mi ordenador personal, la máquina de la UOC y el ordenador Mare Nostrum. Para instalar Hadoop 1.2.1 en mi máquina, voy a los repositorios de Apache en la página web:

https://archive.apache.org/dist/hadoop/core/hadoop-1.2.1/

Seleccionamos el archivo **hadoop-1.2.1-bin.tar.gz** y lo copiamos y descomprimimos en el directorio de nuestra elección. En esta misma página tenemos archivos **rpm** y **deb** por si queremos descargar los paquetes de instalación en lugar de los binarios.

Una vez descomprimido el archivo, tendremos un nuevo directorio denominado **hadoop-1.2.1**, dentro del que podemos ver el subdirectorio **bin** que, como sucede en Linux, es donde tenemos los ejecutables. Podemos ver, asimismo, un archivo

denominado **Hadoop-examples-1.2.1.jar** que contiene una serie de ejemplos y que vamos a emplear para ver si nuestra instalación funciona correctamente.

Llevar a cabo esta prueba es tan simple como ir al directorio hadoop-1.2.1 y ejecutar:

```
Hadoop-1.2.1$ bin/hadoop jar hadoop-examples-1.2.1.jar pi 8 20
```

Este ejemplo estima el valor de Pi, aunque no entraré en los detalles de los parámetros necesarios para su ejecución. Una vez ejecutado este comando, Hadoop nos ofrecerá una gran cantidad de información por pantalla, aunque el nivel de locuacidad de esta salida puede modificarse en la configuración de Hadoop.

El método de instalación tanto en la máquina de la UOC como en el Mare Nostrum III es muy similar. La principal diferencia es que en ninguno de los dos casos se nos permite efectuar conexiones al exterior. Es por ello que deberemos descargar los archivos tar.gz en nuestra máquina y copiarlos en dichos recursos empleando el comando scp de Unix.

A.5 - Instalación de Hadoop 2.7.1

Para instalar la versión 2.7.1 de Hadoop debemos seguir los mismos pasos que en el caso anterior. Accedemos a la página siguiente de los repositorios de Apache:

https://archive.apache.org/dist/hadoop/core/hadoop-2.7.1/

y descargaremos el archivo **hadoop-2.7.1.tar.gz**. Una vez más, lo descomprimiremos en el directorio que creemos a tal efecto y llevaremos a cabo una prueba para comprobar que la instalación y el funcionamiento del programa son correctos.

Para realizar la prueba, iremos al directorio donde está instalado hadoop y ejecutaremos el comando siguiente en la consola:

```
$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.6.0.jar
wordcount input output
```

Hadoop abrirá todos los archivos existentes en el directorio input y contará cuántas veces aparece cada palabra. El programa WordCount no es muy sofisticado, e interpretará como palabras diferentes *"casa"*, *"casa,"* y *"casa."*, pero para probar el correcto funcionamiento del sistema es más que suficiente.
El resultado de la ejecución se escribe en un archivo de texto dentro del directorio output. Este directorio no debe existir al ejecutar el programa porque si no obtendremos un error.

El archivo que he empleado para esta prueba no es otro que el README.txt que aparece en el archivo de instalación de Hadoop.

Como ya he comentado en el apartado anterior, la instalación y prueba de esta versión de Hadoop sería prácticamente equivalente tanto en la máquina de la UOC como en el MN, con la salvedad de la carga del módulo de JDK en el MN.

A.6 - Instalación de Spark 1.4.1 con Hadoop 2.6.0

A.6.1 – Justificación de la elección

Podemos ejecutar Spark de múltiples formas, incluyendo un modo local sobre los archivos de nuestro sistema de archivos particular. Sin embargo, Spark es un motor de proceso de Big Data y sería un poco absurdo emplearlo para este fin. Lo lógico es ejecutar Spark sobre un clúster, para lo cual precisamos de un sistema de archivos distribuido. Puesto que no sólo vamos a familiarizarnos con el HFDS de Hadoop para la elaboración de este trabajo, sino que es un sistema de archivos ampliamente empleado (Google, Facebook, Yahoo,…) parece lógico emplear dicho sistema para ejecutar Spark en el presente trabajo.
Es por ello que he seleccionado el archivo **spark-1.4.1-bin-hadoop2.6.tgz** de los repositorios de Apache para este proyecto.

Por otro lado, Spark precisa de un gestor de clústeres para funcionar en modo distribuido. Aunque ya viene con uno, denominado Standalone Scheduler, Spark puede funcionar sobre una variedad de ellos, como Apache Mesos y Hadoop YARN.
Aunque en principio mi intención es trabajar con el Standalone Scheduler, puesto que es un gestor de clústeres básico y las aplicaciones a desarrollar para Hadoop 2 van a correr sobre YARN, he seleccionado la distribución sobre un Hadoop actual, en lugar de las que corren sobre Hadoop 1, para tener la posibilidad de, llegado el caso (o la tentación), ejecutar Spark sobre YARN.

A.6.2 – Proceso de instalación y prueba

El proceso no es diferente de lo explicado hasta ahora. Primero, vamos a la página de Apache:

https://archive.apache.org/dist/spark/spark-1.4.1

y descargamos el archivo **spark-1.4.1-bin-hadoop2.6.tgz**. Posteriormente, lo copiamos y descomprimimos en el directorio de nuestra elección.
Para efectuar una prueba de su correcto funcionamiento, abrimos una terminal, vamos al directorio de instalación de Spark y ejecutamos el comando siguiente:

```
$ bin/run-example org.apache.spark.examples.SparkPi 8
```

Al igual que en los ejemplos que hemos ejecutado para probar las diferentes versiones de Hadoop, esta aplicación calcula un valor aproximado de pi. El parámetro final es el número de hilos que queremos que cree el programa a la hora de efectuar el cálculo. El número 8 seleccionado es tan bueno como cualquier otro, aunque lo más lógico es que coincida con el número de hilos simultáneos que puede ejecutar nuestro procesador.

Una diferencia con respecto a Hadoop es que no se nos proporcionan métricas al final

de la ejecución. Para poder obtener datos de la ejecución de Spark, debemos tener en marcha el servidor de datos históricos de Spark.

A.7 - Conclusiones

El trabajo descrito a lo largo de este capítulo no presenta una gran complejidad técnica y puede desarrollarse de forma fluida y sin grandes contratiempos. Sin tener conocimientos previos de Hadoop o Spark y siguiendo la documentación de las páginas web correspondientes, podemos tener un sistema funcional y comprobar su correcta instalación en muy poco tiempo.

No obstante, si queremos llegar a una comprensión más profunda de qué estamos haciendo y porqué, necesitamos dedicar bastante más tiempo a la documentación y efectuar unas pruebas más completas, aunque no estoy diciendo nada no aplicable a cualquier faceta de nuestra vida.

He obviado en este caso, como haré en capítulos posteriores, requisitos previos a la instalación del entorno de trabajo, como por ejemplo el tener instalado el ssh y crear una clave pública para que, al ejecutar el sistema en modo pseudo distribuido no tengamos que introducir claves, o crear un script para cargar las variables de entorno adecuadas antes de empezar a trabajar. Si no detallo más las explicaciones es porque considero que es algo que se nos exige saber llegados al punto de realizar el trabajo de fin de grado, aunque son procesos necesarios y que consumen un cierto tiempo.

El trabajo hasta ahora ha sido didáctico, a la par que entretenido. No es lo mismo ver un sistema distribuido en la teoría que trabajar con uno, y ya en este estadio tan temprano del trabajo ya empiezo a ver las relaciones entre la arquitectura a nivel teórico y su implementación.

A.8 - Bibliografía

La bibliografía empleada a lo largo de todo este trabajo es amplia, y ha requerido un importante trabajo de documentación y búsqueda. Los títulos que aparecen a continuación se pueden considerar la bibliografía básica. En el resto de capítulos se incluye bibliografía complementaria, además de ésta, que ha sido necesaria para labores más específicas.

- Turkington, Garry. Hadoop Beginner's Guide. Birmingham: Packt, 2013.
- Holmes, Alex. Hadoop in Practice. 2ª ed. Shelter Island, NY: Manning, 2015.
- Karau, Holden; Konwinski, Andy; Wendell, Patrick; Zaharia, Matei. Learning Spark: Lightning Fast Data Analysis. Sebastopol, CA: O'Reilly, 2015.
- Parsian, Mahmoud. Data Algorithms: Recipes for Scaling up with Hadoop and Spark. Sebastopol, CA: O'Reilly, 2015.
- Documentación oficial de Apache Hadoop desde:

https://hadoop.apache.org/docs/stable/

- Documentación oficial de Apache Spark desde:

  http://spark.apache.org/documentation.html

Anexo B

Notas sobre el código adjunto al proyecto

En el directorio de código adjunto al proyecto nos encontramos con siete archivos Java que paso a explicar brevemente:

HadoopLog
Aplicación de procesamiento de archivos de registro de servidores para MapReduce. Como ya comento en el apartado correspondiente, la elección del nombre es poco afortunada ya que es una aplicación para MapReduce, pero la empecé a desarrollar cuando aún no diferenciaba entre Hadoop y MapReduce y creía que Spark era un sistema aparte, no una parte de Hadoop 2. Como quería que el proyecto también reflejase la evolución de mi aprendizaje, decidí dejar el nombre y también el de HadoopIter.

HadoopIter
Aplicación que implementa un algoritmo iterativo para MapReduce. El algoritmo elegido es el método de Jacobi para la resolución de sistemas de n ecuaciones con n incógnitas. Hace un uso intensivo del HDFS lo que compromete su rendimiento, que es lo que se pretende mostrar. La solución se muestra en el archivo guess, que es la entrada con la estima inicial.

SparkLog
Aplicación de tratamiento de archivos de registro de servidores para Spark. Es equivalente a SparkLog y su salida es igual, salvo por leves diferencias del formato que presenta Spark frente a MapReduce.

SparkIter
Aplicación que implementa el método de Jacobi de resolución de sistemas de n ecuaciones con n incógnitas para Spark. A diferencia de HadoopIter, la salida se ofrece por pantalla para hace el mínimo uso del HDFS y tratar de que el "speedup" sea lo mayor posible.

MRGen
Aplicación que ejecuta cierto número de aplicaciones sobre MapReduce en función de los parámetros de entrada. Está basada sobre HadoopLog y es un simple bucle que va generando trabajos de MapReduce.

MRTrafGen
Máster de generación de tráfico para MapReduce. Creado para HadoopIter, lanza diferentes instancias de Hadoop en función de los parámetros de entrada. Fue la última en construirse, tras comprobar con SparkTrafficGen que era posible y que merecía la pena el esfuerzo.

SparkTrafficGen
Aplicación de generación de tráfico genérico para Spark. Ejecuta diferentes archivos JAR sitos en diferentes directorios de entrada, deducidos a partir de los parámetros de entrada. El número de iteraciones es el mismo para todos los JAR. Quedó en el tintero una mejora en la que en función de un archivo de configuración XML se pudiesen determinar diferentes números de iteraciones para los diferentes JAR. Este archivo XML hubiera eliminado la necesidad de parámetros de entrada para la aplicación.