UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

Escuela Técnica Superior de Ingeniería

Extendiendo una plataforma Big Data multilenguaje para su aplicación a la Bioinformática

Autor:
Gonzalo Muiño Argüelles

Directores:
Juan Carlos Pichel Campos
Tomás Fernández Pena

Máster Universitario en Tecnologías de Análisis de Datos Masivos: Big Data

Julio 2021

Trabajo de Fin de Máster presentado en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela para la obtención del Máster Universitario en Tecnologías de Análisis de Datos Masivos: Big Data

Extendiendo una plataforma Big Data multilenguaje para su aplicación a la Bioinformática

Gonzalo Muiño Argüelles
CiTIUS, Universidade de Santiago de Compostela
gonzalo.muino@rai.usc.es

***Abstract*— En los últimos años, la cantidad de información digital recopilada se ha incrementado de forma significativa en numerosos campos como la biología, la física, la economía y la medicina, dando lugar a lo que llamamos la era del Big Data. Tal ha sido este crecimiento que se han creado nuevos frameworks específicos para gestionar el almacenamiento y posterior análisis de toda esta información y que facilitan el desarrollo de aplicaciones para el procesamiento de estas cantidades masivas en un tiempo razonable haciendo uso de clusters de computadores. Entre estos frameworks destaca Apache Spark que, a pesar de ser una de las herramientas más utilizadas del ámbito de Big Data, tiene limitaciones considerables, entre las cuales destaca la necesidad de hacer uso de determinados lenguajes de programación como Scala, Java, Python o R, requiriendo un esfuerzo significativo de conversión para poder aplicarlo a programas escritos en otros lenguajes de programación. Para solventar este problema se ha desarrollado Ignis, un nuevo framework Big Data que, a través del uso de RPC's, permite la utilización de múltiples lenguajes de programación para conseguir la utilización del paradigma map-reduce. En este trabajo realizaremos una comparación entre Spark e Ignis, analizando el esfuerzo necesario para realizar el portado de aplicaciones de Spark a Ignis, comparando su escalabilidad y realizando una valoración final de ambos frameworks. Nos hemos centrado en aplicaciones de bioinformática, en concreto de genética, ámbito en el que, en los últimos, han aparecido soluciones basadas en tecnologías Big Data para resolver el problema del procesamiento de la enorme cantidad de datos de genoma proporcionados por los secuenciadores de última generación.**

I. INTRODUCCIÓN

Estamos viviendo en una era donde el problema ya no es la falta de información, sino todo lo contrario. Gracias a internet y a la creación e instalación de millones de sensores [1], la generación y recogida de información digitalizada se ha disparado de tal forma que se necesitan herramientas nuevas para poder analizar toda esta cantidad de información. Diversos campos como la biología, la física, la economía y la medicina están viviendo una revolución gracias a esta cantidad de datos, permitiendo el desarrollo de nuevas teorías y la computación de algoritmos antes prohibitivos por falta de *inputs*. No obstante, esto acarrea sus propios problemas como la recogida de esta información, su almacenamiento, la detección de datos erróneos o sobrantes y la dificultad de su procesamiento. Este último, en particular, se ha convertido en el cuello de botella de muchas aplicaciones, necesitando un tiempo todavía muy elevado para poder procesar toda esta información, incluso utilizando clusters con un número elevado de nodos de cómputo.

En el campo de la genética en concreto, las nuevas tecnologías, como los secuenciadores de nueva generación o *Next-Generation Sequencing (NGS)* [2], permiten la obtención de secuencias de ADN a un precio y tiempo reducido. Esta área está pasando por una revolución gracias al procesamiento de esta información, lo que da lugar a nuevos campos como la **filogenómica**, que estudia la evolución de especies utilizando el análisis de ADN, o la **genética médica**, que, a través de la obtención del genoma completo de una persona por menos de 1.000 €, permite una mayor precisión en la detección de enfermedades, teniendo en cuenta el ADN del paciente; en el mejor de los casos puede, incluso, predecir determinadas enfermedades como el cáncer o el Alzheimer, permitiendo tratamientos menos invasivos y con mayor tasa de éxito debido a su detección temprana.

Para la gestión de toda esta información es posible utilizar herramientas de Big Data como Apache Hadoop [3], Apache Flint [4], Google BigQuery [5] y Apache Spark [6]. Este último destaca por su capacidad de procesar enormes cantidades de información en centros de computación, superando a Hadoop y otros frameworks en popularidad y rendimiento. Es por ello que existen numerosas herramientas que la usan para solventar distintos problemas de genética como, por ejemplo *PASTASpark* [7], *SparkBWA* [8], *BiSpark* [9], *Falco* [10], y muchas más.

Pero Spark no está exento de problemas y una de sus limitaciones más importantes es la necesidad de utilizar lenguajes específicos basados en la JVM, teniendo soporte solamente para Java [11], Scala [12], Python [13] y R [14]. El uso de Python en particular también es castigado con una degradación en el rendimiento, debido al uso de pipes para la comunicación con Spark [15]. Esto supone un desafío, ya que muchas de las herramientas existentes de análisis que no hacen uso de frameworks Big Data están implementadas en otros lenguajes como C/C++ o Fortran [16], necesitando un gran esfuerzo de conversión de estos a aquellos soportados por Spark para poder hacer uso de la misma.

Con la intención de eliminar esta limitación, se ha creado en el CiTIUS **Ignis** [15], un nuevo framework Big Data que, por su arquitectura, posibilita el uso de cualquier lenguaje de forma nativa para asegurar que no se pierda ningún control sobre como se deben tratar los datos, soportando incluso herramientas multithreading a nivel local y de esta forma optimizar más el procesamiento de los datos. Este desacople

entre lenguajes se consigue a través de *Remote Procedure Calls (RPC)*, realizando el envío de información a través de **Apache Thrift** [17], con un overhead mínimo para las llamadas entre los distintos componentes, lo cual conlleva a una mejora en el rendimiento con respecto a Spark.

Para poder asegurar esta mejora y también para poder cuantificarla es necesaria la realización de estudios comparativos entre Spark e Ignis. Hasta este momento, Ignis solamente ha sido analizado con benchmarks pequeños que implementan algoritmos del ámbito matemático muy estudiados para mostrar el potencial de la herramienta. En este trabajo se comparará el rendimiento de aplicaciones de código abierto del ámbito de la bioinformática, mas concretamente herramientas que procesan información genética. El análisis de la mejora de rendimiento es esencial en este tipo de herramientas para poder justificar su uso frente a otras alternativas, pero no es el único aspecto relevante ya que se debe también tener en cuenta la usabilidad, adaptabilidad y seguridad de la aplicación.

Esta comparación se realizará portando distintas herramientas de código abierto que hagan uso de Spark a *Ignis*, obteniendo tiempos de ejecución de distintas ejecuciones con varios datasets en cada programa. Se utilizarán herramientas de distintas complejidades, con intención de variar el tipo de manipulación de información que realicen y de obtener también tiempos de ejecuciones desde segundos hasta varias horas.

En este trabajo comenzamos introduciendo Spark y MapReduce detallando sus componentes, su funcionamiento así como sus ventajas y limitaciones frente a otras herramientas en la sección 2. Sección 3 presenta a Ignis y explica como soluciona estas limitaciones de Spark. Hacemos una comparación más detallada de ambas herramientas en sección 4, listamos las aplicaciones a través de las cuales se va a realizar la comparación en sección 5. En sección 6 mostramos las dificultades que han surgido a lo largo de la realización de este trabajo, comparamos el rendimiento de ambas aplicaciones en sección 7 y finalmente sacamos las conclusiones oportunas del trabajo realizado, donde también detallaremos futuros trabajos a realizar, en la sección 8.

II. Apache Spark

Desarrollado en la universidad de California, Berkley por el **AMPLab** y lanzado el 26 de mayo de 2014, Spark [6] es un framework Big Data de código abierto para el procesamiento de datos masivos que hace uso de un cluster de computación siguiendo el modelo MapReduce [18], propuesto por Google. En este modelo, la información se representa a través de parejas clave-valor y la ejecución se divide en dos partes:

- **Map**: La información es manipulada de tal forma que la computación de una pareja clave-valor es independiente de la de cualquier otra pareja (incluyendo aquellas que tengan la misma clave), lo que permite un nivel muy alto de paralelización. La salida de esta computación debe ser de nuevo parejas de clave-valor. Habitualmente la parte map es ejecutada al comienzo.
- **Reduce**: Al igual que en el Map, la entrada y la salida son parejas clave-valor, pero a diferencia de lo que sucede en Map, en Reduce la independencia de cálculo solo se da entre parejas donde la clave es distinta. Aquellas que tengan la misma clave deben ser ejecutadas por el mismo nodo. Habitualmente la parte reduce es ejecutada al final para la obtención final de los resultados.

La primera implantación de código abierto de éxito de este modelo no fue Spark, sino Hadoop. Sin embargo, Spark fue propuesto como una alternativa a Hadoop más eficiente para la manipulación iterativa de información. A diferencia de Hadoop, donde cada resultado se almacenado en el disco duro, Spark representa su información a través de **Resilient Distributed Datasets** o RDDs, colecciones de datos distribuidas y que se intentan mantener en RAM, reduciendo los accesos a disco pues se copian al mismo solo si es imprescindible. Spark también permite la reutilización continua de RDDs en lugar de requerir un cálculo cada vez que se necesite y también hace una ejecución inteligente utilizando grafos acíclicos dirigidos (*Directed Acyclic Graph* o DAG) para determinar el orden óptimo de las tareas y también para determinar aquellas tareas que no sean necesarias. Esto da lugar a que Spark sea hasta 100 veces más rápido que Hadoop en algunas aplicaciones. A mayores, Spark ofrece módulos que proporcionan herramientas específicas para Machine Learning, procesamiento en streaming, procesamiento de grafos e incluye una integración de SQL.

A pesar de que puede funcionar sin ninguna otra herramienta, habitualmente Spark se ejecuta sobre *Hadoop Distributed File System* (HDFS) [19] y *Yet Another Resource Negotiatior* (YARN) [20]:

- **Hadoop Distributed File System** o **HDFS**: Diseñado originalmente para Hadoop, se encarga de la gestión del almacenamiento de la información. Divide los ficheros en bloques que son enviados a los distintos nodos del cluster y que están replicados un número determinado de veces para asegurar que no se produzcan pérdidas en caso de que uno de los nodos falle. Toda esta gestión es transparente al usuario, para el cual HDFS simplemente es una carpeta más donde tener su información. Spark hace uso de este reparto de información para determinar el lugar de ejecución óptimo de los programas, evitando movimiento de la información.
- **Yet Another Resource Negotiatior** o **YARN**: También diseñado originalmente para Hadoop 2, se ocupa de gestionar los recursos al determinar qué nodos debe asignar a cada uno de los trabajos recibidos, así como haciendo un seguimiento de estos trabajos y del estado de los nodos. En esencia es el que se encarga de realmente lanzar los trabajos sobre el cluster, permitiendo la ejecución simultánea de varios trabajos sin que estos se disputen los recursos.

Las aplicaciones ejecutadas en Spark se dividen en dos partes:

- **Driver**: Parte del código que controla qué tareas deben ejecutar los distintos nodos. No realiza ninguna manipulación directa de la información, sino que determina qué funciones deben ser ejecutadas en los distintos nodos para obtener nuevas parejas clave-valor. Se ejecuta solamente en un nodo.
- **Ejecutor**: Parte del código que se ejecuta en los distintos nodos del cluster. Se ejecuta varias veces simultáneamente en paralelo y son los que se encargan de realizar la manipulación de los datos, procesándolos de forma independiente entre ellos.

Spark realiza la manipulación de la información utilizando estas dos partes. El nodo donde se ejecuta el driver es el que envía al manager la información necesaria para esta ejecución, que se encarga de enviar a los distintos nodos trabajadores las tareas que son determinadas por el programa driver. Las distintas tareas que se generan se intentarán realizar en el mismo nodo en que se encuentre la información como tal físicamente, determinado a través de HDFS. Si esto no es posible, se realiza un envío de datos al nodo donde se necesiten. Algunas tareas también requieren un envío de información por su propia naturaleza (*parallelize()*, por ejemplo).

Las tareas tareas se dividen en dos tipos: Transformaciones y Acciones. Las transformaciones son aquellas cuyo resultado es otro RDD, mientras que las acciones requieren o bien movimiento de la información (*collect()*, por ejemplo) o escritura de información al disco duro.

Spark hace uso de *Lazy Evaluation*, o evaluación vaga, o sea que una transformación no se ejecute hasta que se haya determinado indispensable. Es decir, si una transformación no da lugar a un RDD que es utilizado por una acción posterior o una transformación wide que fuerza la carga de datos y la ejecución de transformaciones anteriores, esta no se va a ejecutar. Por lo tanto, una ejecución de Spark sigue los siguientes pasos:

1) Solicitud y asignación de recursos (Nodos, CPUs, Memoria, entre otros).
2) Arranque del Driver. Determinación de las transformaciones y acciones a realizar. Si una transformación no es necesaria, no será ejecutada en ningún momento.
3) Ejecución de estas transformaciones y posterior acción.
4) Volver al Driver para continuar su ejecución. Este ahora puede volver a determinar que es necesario la ejecución de nuevas transformaciones y acciones o puede finalizar todo.
5) Liberación de los recursos asignados y finalización del programa en el cluster.

Haciendo uso de todo lo que hemos dicho, Spark permite la ejecución en paralelo de los ejecutores en un cluster, proporcionando tiempos de ejecución reducidos cuando trabajamos con datasets sumamente grandes, dando lugar a que sea el framework Big Data más utilizado. Sin embargo, Spark tiene también desventajas. Una de ellas es la necesidad de utilizar lenguajes basados en la JVM, como Scala y Java, Python o R. A pesar de que el soporte es total para estos (no existe ninguna función disponible en Java o Scala que no esté disponible en Python o R), utilizar Python y R produce una degradación añadida del rendimiento [42]. Esta degradación está causada por la necesidad de pasar la información de Python o R a Java, el lenguaje sobre el cual se ejecuta Spark. Este intercambio de información se realiza transformándola a *string*, para luego volver a ser parseada a los tipos de datos nativos del lenguaje al cual se está haciendo el paso. Nótese que esta transformación debe ser repetida dos veces cada vez que se realiza un envío de información, una de Python a Java y luego otra en sentido contrario. Esta transformación a *string* es laboriosa cuando tratamos con tipos o estructuras de datos complejas como *hashmaps* o árboles no binarios. Es por ello que existe también un interés especial en tener un framework Big Data que permita el uso de Python y R sin esa pérdida de rendimiento, además de dar soporte a otros lenguajes no soportados por Spark como C/C++.

III. Ignis

Para solucionar estas limitaciones, en el CiTIUS se ha creado Ignis [15]. Al igual que Spark, Ignis también hace uso del modelo Map-Reduce y ofrece las mismas ventajas que Spark frente a Hadoop, así como otras características como el almacenamiento de la información intermedia en memoria en lugar del disco duro, la determinación de transformaciones innecesarias, la división en Driver y ejecutor de las aplicaciones, etc.

Sin embargo, una de las diferencias más significativas es la posibilidad de utilizar lenguajes distintos en el driver y en el ejecutor. No solo eso, sino que Ignis posibilita la utilización de ejecutores escritos en varios lenguajes. Esto permite al programador adaptar cada transformación y acción al lenguaje más apropiado, permitiendo un mejor uso de los recursos disponibles. Ignis consigue esto a través de Apache Thrift, una herramienta que posibilita el uso de Remote Procedure Calls (RPC) para conseguir este desacople. Destacar que Spark también hace uso de Thrift para realizar llamadas entre los distintos componentes, pero el mayor desacople de los componentes de Ignis permite este uso de varios lenguajes. Veremos ahora cuales son estos componentes y como consigue Ignis esta independencia de lenguajes. Ignis consiste en cuatro módulos independientes:

- **Driver**: Es el componente que se encargará de la ejecución del código driver. No realiza ninguna manipulación directa de la información. A diferencia de Spark, no necesita estar implementado en Java, Scala, Python o R, sino que puede ser implementado en cualquier lenguaje que esté soportado en Ignis (actualmente solo Python y C/C++, pero es fácilmente expandible a otros lenguajes a través de la creación de librerías). Delega todo el trabajo en el módulo de backend, o sea, sirve de interfaz para las funciones implementadas en el módulo de backend, permitiendo la inclusión de nuevos lenguajes sin necesidad de

re-implementar toda la lógica backend, ya que sólo es necesario definir funciones de llamada al backend y la definición del paso de la información a través de Thrift.
- **Backend**: Sirve como pieza intermedia que traduce las peticiones recibidas por el driver en tareas más concretas que se van a ejecutar en el módulo ejecutor. Se encarga también de solicitar recursos al módulo de Manager, de realizar el intercambio de información entre los distintos ejecutores y de recuperarse de pérdidas de información, errores de ejecución o caídas de los distintos nodos.
- **Ejecutor**: Módulo que se encarga de la ejecución de las tareas. La inclusión de un nuevo lenguaje de ejecución requiere la implementación de este módulo en ese lenguaje concreto. Destacar que el soporte de lenguajes en Driver y Ejecutor no tiene por que ser el mismo, permitiendo la adaptación independiente de estos.
- **Manager**: Parte intermedia entre el backend y el ejecutor. Se encarga de lanzar los ejecutores y de asegurarse de que sigan vivos.

Todas los módulos se ejecutan dentro de imágenes Docker [21]. Existe un quinto componente de la arquitectura, el *Docker Resource Manager* que se encarga de lanzar las distintas imágenes Docker, realizar la asignación de recursos físicos a estas imágenes y asegurarse que las imágenes sigan vivas.

En Ignis, el almacenamiento de la información se realiza a través del acceso a una carpeta común a todos los ejecutores. No tiene soporte para HDFS, pero si permite la escritura en paralelo en esta carpeta facultando la paralelización del almacenamiento de la información, como también permiten Spark y Hadoop.

Como podemos ver, Ignis y Spark presentan arquitecturas similares. La inclusión de un módulo adicional permite a Ignis independizar el Driver y el Ejecutor, lo cual posibilita el uso de un mayor número de lenguajes, y también el no sufrir un deterioro del rendimiento por utilizar lenguajes no JVM.

Procederemos entonces con la comparación de estos dos frameworks, destacando primero las diferencias más significativas a nivel de programación.

IV. Spark vs Ignis

Vamos a realizar primero una comparación directa de ambos frameworks. Lo primero que compararemos es el número de lenguajes soportados. Spark que ya tiene 7 años, tiene soporte solamente para lenguajes JVM (Java y Scala), Python y R. Para estos últimos, como ya se indicó, se produce una degradación en el rendimiento. Ignis, por otra parte, a pesar de estar todavía en fase de desarrollo, tiene soporte ya para Python y C/C++, con soporte parcial para lenguajes JVM y Go [22]. El uso de Apache Thrift para la comunicación inter-modular permite la rápida adaptación de las herramientas a otros lenguajes sin necesidad de reimplementar toda la lógica y, lo que es más importante a largo plazo, sin un deterioro en el rendimiento respecto a lenguajes ya implementados. Esta adaptación se efectúa en dos fases, una para el ejecutor y otra para el driver.

El envío de los trabajos al cluster se realiza de forma idéntica, teniendo Spark el comando spark−submit e Ignis su equivalente con ignis−submit. En ambos casos podemos definir configuraciones ya en el momento de ejecución a través de sendas opciones si no queremos que sean definidos en el código.

Otra diferencia significativa es la dificultad de ambos lenguajes. La API de Ignis fue diseñada de tal forma que un usuario familiarizado en el uso de Spark no tendrá problema en adoptar Ignis. Para ello, se les han dado a las distintas funciones para manipular los datos nombres idénticos a las encontradas en Spark como map(), reduce() o saveAsTextFile(). Sin embargo, Ignis sí tiene más complejidad debido a la necesidad de declarar el lenguaje de los ejecutores. Comentaremos un ejemplo directo implementando en Spark e Ignis un **Wordcount** (ver Figura 1), el considerado como *Hello World* del Big Data.

Como vemos en la figura 1, la definición de las funciones, así como la aplicación de dichas funciones a los conjuntos de datos, es idéntica en ambos frameworks. También podemos observar cómo la definición de las propiedades es muy similar, sustituyendo Ignis un diccionario por una clase propia de Spark. La única diferencia significativa que se puede encontrar es en la necesidad de crear el trabajador en Ignis , usando el método IWorker(). Esto es necesario debido a la posibilidad de definir trabajadores de distintos lenguajes. Más adelante veremos un ejemplo de Ignis donde utilizamos dos trabajadores con funciones definidas en distintos lenguajes.

Destacar que ambos ejemplos se pueden simplificar significativamente a través de llamadas sucesivas de las funciones de manipulación de datos en lugar de guardar los resultados intermedios. A su vez, tanto Spark como Ignis soportan la utilización de funciones lambdas sin ninguna modificación a su rendimiento.

Podemos afirmar, pues, que el portado de aplicaciones de Spark a Ignis es muy sencillo. Con la excepción de la necesidad de definir el trabajador, que en este caso añade una línea de código, no tenemos que realizar ninguna modificación adicional. Esto facilita el paso de herramientas que ya estén utilizando Spark a Ignis.

Centrándonos ya más en las diferencias, Ignis permite la utilización de varios ejecutores de distintos lenguajes de programación en una misma ejecución. Consideremos el ejemplo de la figura 2, en donde ejecutamos una función con un ejecutor escrito en Python y otra función con un ejecutor en C++.

Como podemos ver, la transición de un ejecutor a otro solamente requiere la utilización de la función importData() para realizar el paso de la información. No es necesario especificar ninguna equivalencia de tipos, el paso de información se realiza de forma automática. Esto nos permite trabajar con múltiples programas, cada uno escrito en su propio lenguaje de programación, para paralelizar su ejecución en un cluster. A efectos prácticos, esto nos

```python
# Versión Ignis

# Definimos las funciones a utilizar
def funcionSplit(linea):
    return linea.split(" ")


def funcionUno(palabra):
    return (palabra, 1)


def funcionSuma(A, B):
    return A + B


# Arrancamos el framework
ignis.Ignis.Start()

# Definimos la configuración
conf = ignis.IProperties()
prop["ignis.executor.image"] = "ignishpc/full"
prop["ignis.executor.instances"] = "4"
prop["ignis.executor.cores"] = "5"
prop["ignis.executor.memory"] = "10GB"

# Obtenemos el entorno de Ignis
cluster = ignis.ICluster(conf)

# Creamos un ejecutor de python
ejecutor = ignis.IWorker(cluster, "python")

# Abrimos el archivo cuyas palabras queremos contar
archivo = ejecutor.textFile("Ruta/Hacia/Archivo")

# Aplicamos el split, quedándonos con las palabras de cada línea
palabras = archivo.flatMap(funcionSplit)

# Pasamos por todas las palabras anotando todas las ocurrencias
ocurrencias = palabras.map(funcionUno)

# Realizamos el conteo de todas las ocurrencias por palabra
sumaTotal = ocurrencias.reduceByKey(funcionSuma)

# Obtenemos la lista de todas las ocurrencias
listaOcurrenciasTotales = sumaTotal.collect()
```

```python
# Versión Spark

# Definimos las funciones a utilizar
def funcionSplit(linea):
    return linea.split(" ")


def funcionUno(palabra):
    return (palabra, 1)


def funcionSuma(A, B):
    return A + B


# Definimos la configuración
conf = SparkConf()
conf.set("spark.executor.instances", "4")
conf.set("spark.executor.cores", "5")
conf.set("spark.executor.memory", "10GB")

# Obtenemos el entorno de Spark
sc = SparkContext(conf)

# Abrimos el archivo cuyas palabras queremos contar
archivo = sc.textFile("Ruta/Hacia/Archivo")

# Aplicamos el split, quedándonos con las palabras de cada línea
palabras = archivo.flatMap(funcionSplit)

# Pasamos por todas las palabras anotando todas las ocurrencias
ocurrencias = palabras.map(funcionUno)

# Realizamos el conteo de todas las ocurrencias por palabra
sumaTotal = ocurrencias.reduceByKey(funcionSuma)

# Obtenemos la lista de todas las ocurrencias
listaOcurrenciasTotales = sumaTotal.collect()
```

Fig. 1: Comparación de códigos de Ignis (izqda.) y Spark (dcha.) para la aplicación Wordcount.

permite combinar herramientas en un pipeline de forma eficaz y sencilla utilizando Ignis, optimizando de forma independiente cada parte y permitiendo centrar nuestro esfuerzo en conseguir esta paralelización, en lugar de tener que centrarlo en re-programar dichas herramientas a un lenguaje común. Spark no ofrece esta posibilidad, ya que requiere que se ejecuten todos los ejecutores en un mismo lenguaje.

Una de las desventajas de la versión actual de Ignis es que solo tiene implementadas las funciones de Spark-Core (el módulo principal de Spark) y SparkSQL [23], careciendo de soporte para las funciones de Spark Streaming [24], MLlib [25] (módulo de machine larning) y GraphX [26] (módulo de grafos). Esta limitación es solo temporal y será corregida en futuras revisiones de Ignis.

V. Aplicaciones

Para comparar y analizar el rendimiento de ambos frameworks es necesario el uso de aplicaciones reales para poder valorar el tiempo de ejecución de ambas versiones. Listaremos aquí las distintas herramientas que vamos a utilizar, su finalidad y demás detalles de las mismas como características de los datos utilizados en su ejecución.

A. Tomatula

Tomatula [27, 28] es una aplicación que permite manejar, a través del uso de Hadoop y Spark, archivos en formato VCF (*Variant Call Format*), permitiendo realizar búsquedas concretas de alelos y también obtener el número total de las frecuencias de cada uno de ellos. Un alelo son variaciones de un mismo gen. Están localizados en la misma posición de un cromosoma y todas las personas tienen 2 alelos del mismo gen, uno heredado del padre y otro de la mdre. Para obtener el número total de las frecuencias de estos, hace una primera transformación del VCF a formato JSON, y luego genera diccionarios que contienen la información dentro de RDDs. Tiene soporte también para determinar la frecuencia de alelos a partir de archivos en formato Apache Parquet, un formato propio de Hadoop y Spark. Con este fin, hace uso de HDFS, posibilitando realizar la carga de forma rápida

```
# Arrancamos el framework
ignis.Ignis.Start()

# Definimos la configuración
conf = ignis.IProperties()
prop["ignis.executor.image"] = "ignishpc/full"
prop["ignis.executor.instances"] = "4"
prop["ignis.executor.cores"] = "5"
prop["ignis.executor.memory"] = "10GB"

# Obtenemos el entorno de Ignis
cluster = ignis.ICluster(conf)

# Creamos un ejecutor de python
ejecutorPython = ignis.IWorker(cluster, "python")

# Abrimos un archivo
archivo = ejecutorPython.textFile("Ruta/Hacia/Archivo")

# Aplicamos una función definida en un archivo .py
salidaPython = archivo.map("Ruta/Hacia/Archivo.py:NombreFuncion")

# Creamos un ejecutor de C++
ejecutorC = ignis.IWorker(cluster, "cpp")

# Importamos la información de python a C++
datosImportados = ejecutorC.importData(salidaPython)

# Aplicamos una función definida en un .so
salidaC = datosImportados.map("Ruta/Hacia/Archivo.so:NombreFuncion")
```

Fig. 2: Código en Ignis para una aplicación con dos ejecutores de lenguajes distintos.

y directa, en vez de tener que realizar el envío del archivo desde el Driver.

Con Tomatula vamos a comprobar como la transición entre Spark e Ignis es inmediata cuando estamos trabajando con herramientas sencillas como esta. No hace uso de ningún tipo de llamada externa a herramientas de terceros, ni tiene ninguna dependencia con librerías externas (más allá de Spark). Sin embargo, veremos cómo incluso en programas muy simples Ignis obtiene mejor rendimiento que Spark, a cambio de lo que es esencialmente ningún incremento en la complejidad del programa. Para esta herramienta en particular, las modificaciones necesarias para el portado son idénticas a las vistas en la figura 1.

B. PPCAS

PPCAS (*Probabilistic Pairwise model for Consistency-based multiple alignment in Apache Spark*) [29, 30], es una aplicación que permite el alineamiento de secuencias de proteínas en formato FASTA a través de comparaciones por parejas utilizando el algoritmo **ProbCons**. El resultado del alineamiento se puede almacenar en numerosos formatos. Cuando realiza las comparaciones entre dos secuencias, PPCAS tiene que comprobar $Longitud(secuencia1) \times Longitud(secuencia2)$ solapamientos, por lo que la complejidad computacional se incrementa exponencialmente cuando incrementamos el número de secuencias y su longitud, situación típico en los sistemas NGS. Estas comprobaciones también son independientes, ya que para cada solapamiento tendremos un porcentaje de alineamiento calculado independientemente de los demás solapamientos, dando lugar a un algoritmo que está perfectamente adaptado a la utilización de herramientas Big Data.

En cuanto a su implementación, PPCAS es una aplicación compleja que hace uso de una librería propia en C para el cálculo de las probabilidades que se incorpora en Python a través del módulo *ctypes* [31]. PPCAS hace también uso de la función broadcast(), que permite el envío de información completa a todos los nodos del cluster, en lugar de hacer un particionado y enviar a cada nodo una partición distinta. En este caso, la información que se envía a todos los nodos es la lista original de las secuencias. En Ignis no es necesario la utilización de broadcast, ya que permite acceder desde todos los nodos a todas las variables que estén en el entorno en el momento que se mandan los trabajos a los ejecutores.

C. PASTASpark

Desarrollado en el CiTIUS, PASTASpark [7, 32] es una herramienta que hace uso de Spark para acelerar parte del procesamiento de PASTA (*Practical Alignments using SATé and TrAnsitivity*), una herramienta para obtener el alineamiento de secuencias múltiples. Utiliza un archivo FASTA y un árbol para realizar este alineamiento; a falta de un árbol genera uno de forma automática.

Destacar que PASTASpark tiene una parte importante que no es paralelizable. Como consecuencia, a diferencia de las demás herramientas descritas aquí, la reducción de tiempo con varios ejecutores o varios cores no será tan pronunciada como en otros, que casi siguen un descenso lineal con el número de ejecutores y cores.

A mayores de las herramientas de bioinformática ya descritas previamente, se utilizaron también otras aplicaciones provenientes de otras área de interés para comparar rendimientos entre ambos frameworks.

D. Summarizer

La herramienta Summarizer [33, 34] pertenece al ámbito del procesamiento del lenguaje natural (PLN), y realiza un análisis semántico latente de un conjunto de reviews de productos en Amazon. Recorre, en primer lugar, el dataset entero para calcular la matriz de frecuencia de términos y posteriormente la matriz TF-IDF. Utilizándola, realiza el análisis semántico para obtener 10 agrupaciones de frases que tengan alguna relación semántica. De esta forma podemos obtener una agrupación que asocie frases positivas sobre un producto o frases que hablen sobre un determinado tipo de producto.

VI. Dificultades del portado de Spark a Ignis

Para este trabajo no se lidió solamente con el la migración *per se* de las herramientas de Spark a Ignis, sino que también hubo que realizar otros cambios para el correcto funcionamiento de Ignis. De entrada, la versión de Spark que se ha utilizado es la 2.2.0 utilizando la versión 2.7 de Python. Por su parte, Ignis hace uso de la versión 3 de Python. Por esta razón ha sido necesario el uso de la herramienta 2to3 [35], permitiendo la conversión de las aplicaciones de

la versión 2.7 a 3 de forma bastante sencilla, con la excepción de PASTASpark.

PASTASpark hace uso de la librería DendroPy [36], en particular, de su versión 3. Sin embargo, esta versión de DendroPy no tiene soporte para Python 3 y la utilización de la herramienta *2to3* no ayuda a solucionar este problema. Por lo tanto, fue necesario el uso de la versión 4 de DendroPy, lo que requirió una refactorización de PASTASpark para utilizar las nuevas funciones de DendroPy. Algunas de las adaptaciones supusieron una simple conversión de datos (realizar el paso de un string a formato Byte), pero otras implicaron un mayor número de modificaciones tanto en el código de PASTASpark como en el de DendroPy. Destacar que este problema no estaba causado por portar PASTASpark a Ignis, sino por la necesidad de implementar PASTASpark en Python 3. Nos encontramos con un problema similar en el módulo *numpy*, utilizado en muchas herramientas, pero en este caso simplemente fue necesario la instalación y el uso de una versión con soporte para Python 2.7 y 3 simultáneamente.

Otra complicación fue la diferencia en el comportamiento entre Spark e Ignis con el uso de ficheros compartidos. En lugar de utilizar HDFS como Spark, Ignis tiene tiene que acceder a los archivos situados en la carpeta `/media/dfs` en la imagen Docker del controlador, necesitando una copia de los ficheros de entrada en esa carpeta. A mayores, fue necesario el almacenamiento de los códigos fuentes y las dependencias en esta carpeta, una limitación no presente en Spark en donde el usuario tiene más libertad para determinar su estructura de archivos.

La utilización de un framework como Ignis que se encuentra en la actualidad todavía en desarrollo conlleva problemas de estabilidad, bugs, falta de documentación y la necesidad de esperar a que se arreglen errores y bugs de herramientas externas. En particular de MPICH [37] que causó problemas significativos en el uso de Ignis con Python, al no permitir el uso de MPI para el envío de información de los nodos al controlador. Se han realizado modificaciones para solventar este problema, sin embargo funciones como reduce, collect y demás que realizan un movimiento de datos desde los trabajadores al controlador siguen siendo problemáticas cuando se quiere transmitir un alto volumen de datos. Como consecuencia no se han podido usar datos de entrada de gran tamaño a la hora de evaluar las aplicaciones Summarizer y PASTASpark, que hacen amplio uso de estas funciones, a mayores de sufrir mayor número de cambios frente a Tomatula o PPCAS.

En Summarizer las modificaciones limitan también el tamaño del dataset, ya que la implementación original hace uso de DataFrames ordenados por clave desde los cuales se obtiene el dato a través de la función lookup(), que trae el dato desde los nodos al controlador. Dado que Ignis carece de esta funcionalidad en su API, ha sido necesario utilizar un collect () y crear un diccionario local para realizar el lookup() de forma local en el controlador, mejorando de forma significativa el tiempo de búsqueda ya que no se tiene que realizar el envío de forma repetida en algunos casos. Dado que esto representa un cambio significativo en el rendimiento no debido al uso de Ignis sino a una reimplementación de la lógica del programa, el mismo cambio se ha realizado en Spark para que las pruebas de rendimiento sean justas.

En el caso de PASTASpark fue necesario realizar más modificaciones relacionadas con la utilización de herramientas secundarias como **Hmmeralign** [38] y **Mafft** [39], para las cuales hubo que hacer una adaptación de la salida de DendroPy 4 para el uso de estas herramientas. PASTASpark también necesitó un cambio en donde una función map(), que era interna a la clase que realizaba el envío del trabajo al controlador de Ignis, tuvo que sacarse a un fichero externo debido a una dependencia con el propio Spark, localizado en `/media/dfs`, que Ignis no encontraba. El fichero externo, libre de dependencias, pudo ser lanzado correctamente y se ha realizado la carga de estas dependencias (necesarias para la función map()) en su interior tras añadir `/media/dfs` al path de búsqueda de dependencias.

Finalmente, se ha hecho una última modificación de PASTASpark debido al tamaño del dataset empleado. A pesar de que se traten de datasets relativamente pequeños respecto a lo que nos podemos encontrar en un caso real de uso de la herramientas, la ejecución de estos, en particular el dataset más grande, es de varias horas. Sin embargo, la parte paralelizable representa una hora del tiempo total en caso de la ejecución con un solo core y solamente 10 minutos en el caso de 16 cores. Para poder comparar con mayor detalle el rendimiento entre Spark e Ignis se van a realizar mediciones solamente de la parte paralelizable de PASTASpark, obviando la parte no paralelizable.

La falta de aplicaciones de Bioinformática que hagan uso de PySpark representa otro problema. Como Ignis no tiene todavía soporte para lenguajes JVM, quedan excluidas herramientas como SparkBWA [8] del pool de aplicaciones disponibles. Dado que Java y Scala son los lenguajes más utilizados con Spark, esto limita el número de aplicaciones disponibles. La falta de implementación de MLlib y de GraphX límita todavía más las aplicaciones.

También fue problemático el proceso de detección de errores y corrección de los mismos. A falta de herramientas de depurado sólidas fue necesario el uso de logs para determinar errores a corregir, tanto a nivel de Ignis como de las aplicaciones que se han probado. Esto conlleva un proceso lento en la detección de estos errores, particularmente cuando el error se debe a una dependencia de terceros, como es el caso de MPICH. Cada modificación significativa de Ignis requería también un elevado tiempo de espera, algunas veces superior a una hora, hasta el despliegue en el cluster.

VII. Resultados experimentales

Todas las ejecuciones se han realizado en el cluster Big Data del CiTIUS. Este consiste de 14 servidores Dell EMC PowerEdge R370, donde cada uno tiene:

- 2 Intel Xeon E5-2630 v4 (2,2 Ghz, 10 cores)

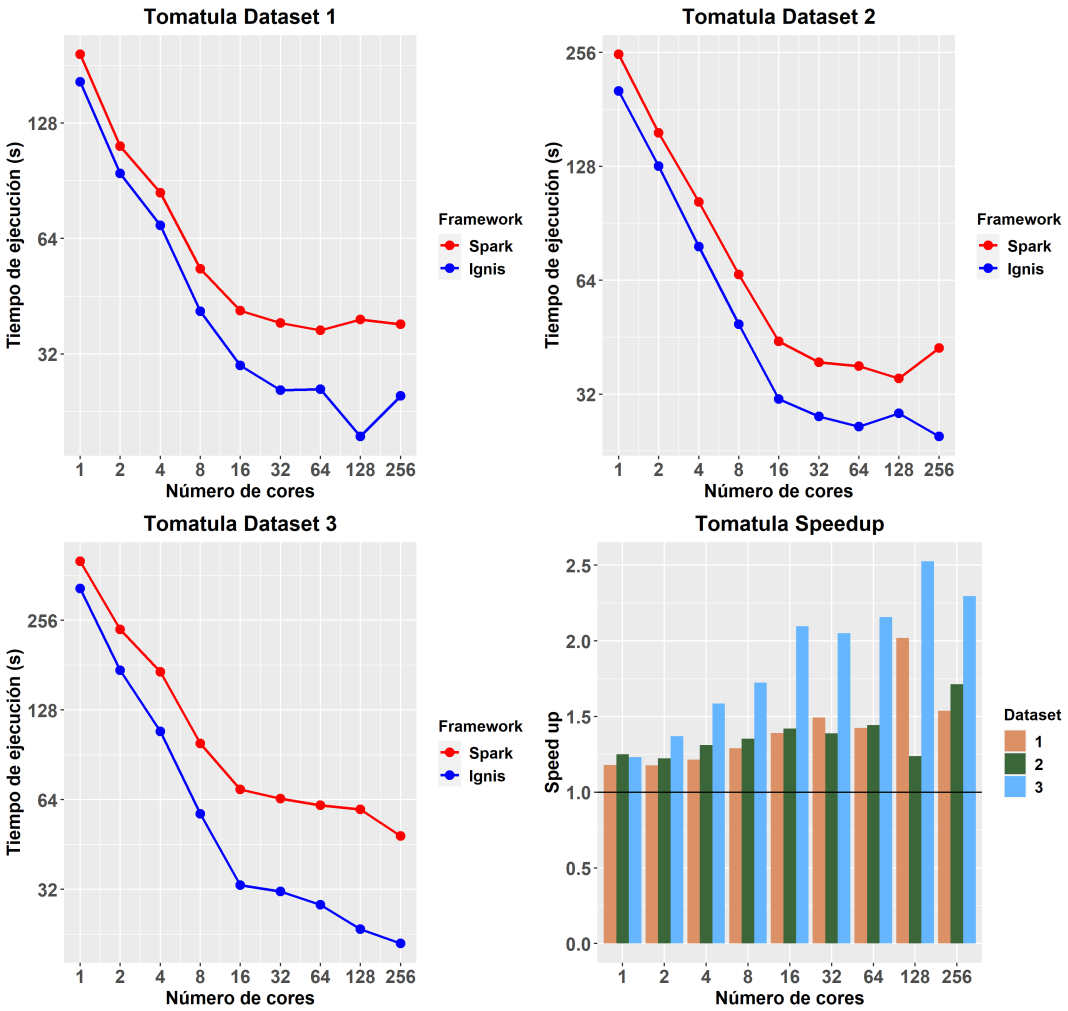

Fig. 3: Análisis de la escalabilidad de Tomatula.

- 384 GB de RAM: 12 × 32 GB RDIMM 2400 MT/s
- 32 TB HDD: 8 × 4 TB 7.2k SATA 6 Gbps en JBOD
- 2 × 10Gb BaseT y 2 × 1Gb BaseT

En nuestras pruebas hemos variado el número de cores usados para estudiar la escalabilidad de las aplicaciones analizadas. Para todas las ejecuciones se ha utilizado la misma cantidad de RAM tanto en Spark como en Ignis. Las comparaciones se muestran en una gráfica para cada dataset con los tiempos de ejecución (en segundos) de Spark e Ignis, además de con una figura mostrando el speedup en función del número de cores y del dataset. El speedup representa la aceleración de Ignis frente a Spark. Es decir, un speedup mayor que 1 representa una mejora de rendimiento de Ignis frente a Spark y un speedup menor que 1 representa lo contrario. Las gráficas de tiempos para las aplicaciones Tomatula y PPCAS son logarítmicas (tanto en el eje X como Y). Los tiempos se han obtenido ejecutando todas las herramientas 5 veces con cada combinación de dataset de entrada y número de cores, obteniendo la media de los tiempos de ejecución como el valor final.

Comenzaremos analizando los resultados de Tomatula, mostrados en la Figura 3. En este caso se han utilizado dos datasets del genoma humano procedentes del National Center for Biotechnology Information [40] [Datasets 1 y 2] así como un dataset del proyecto 1000 Genomes [41]. Los tres son archivos VCF con un tamaño de 4, 13 y 17 GB, respectivamente.

Como se puede observar, para cualquier dataset y tamaño, Ignis es siempre más rápido que Spark. Sin embargo la diferencia varía en función de los datasets y del número de cores, siendo el speedup mayor con los datasets más grandes y cuando se utiliza un mayor número de cores. En particular, con el dataset más grande el speedup se incrementa notablemente a medida que se incrementa el paralelismo, siendo Ignis hasta 2.5× más rápido que Spark.

Continuando con la aplicación PPCAS, el dataset utilizado viene ya incluido en el repositorio de la herramienta. Se tratan de tres ficheros FASTA con secuencias genéticas, en donde el primero tiene un peso de 12 KB y 212 líneas, el segundo un peso de 104 KB y 2.012 líneas y el tercero tiene un peso de 514 KB y 10.012 líneas. Destacar que en este caso la complejidad del problema se dispara debido a la cantidad de comprobaciones que se deben realizar al comprar todas las cadenas. De hecho, aun teniendo un dataset pequeño con

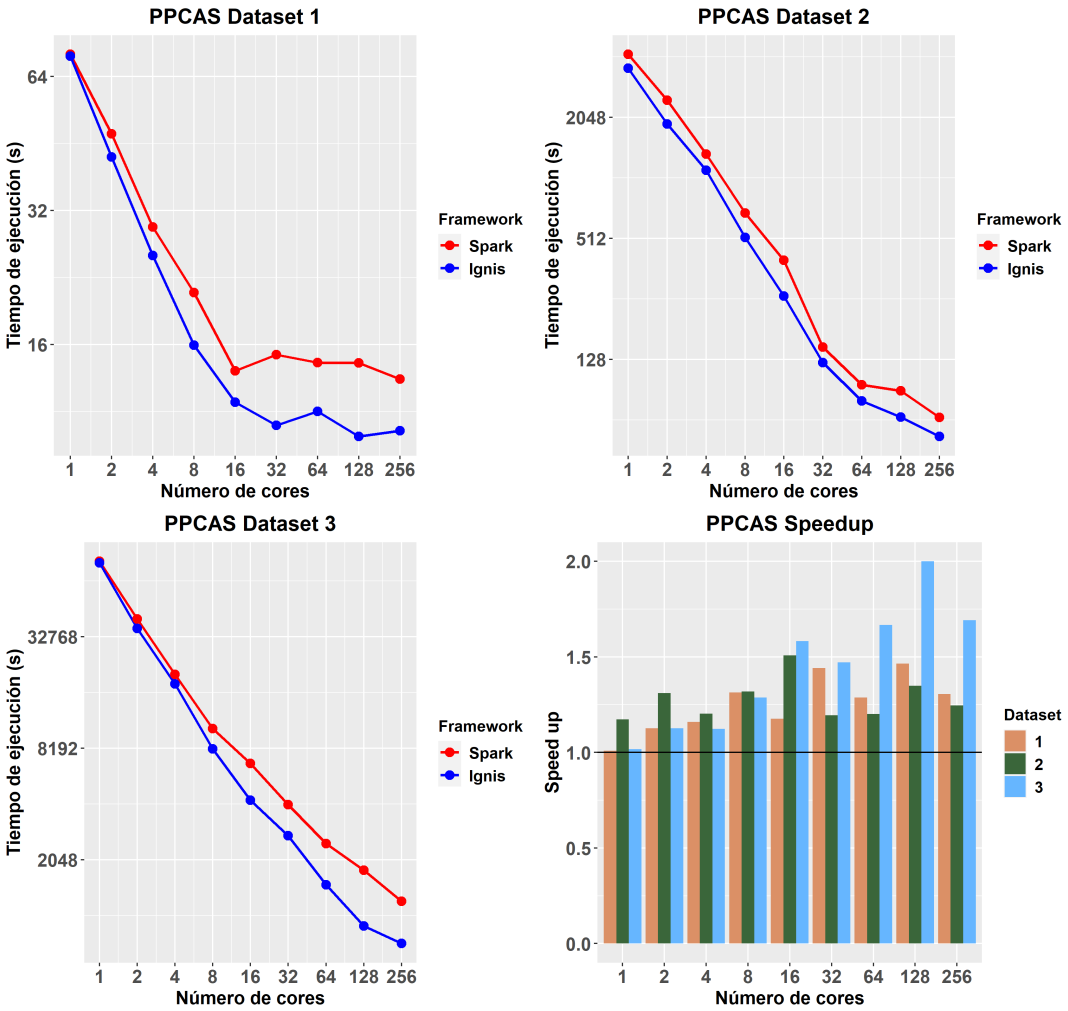

Fig. 4: Análisis de la escalabilidad de PPCAS.

poco más de 500 KB, el tiempo de ejecución es de varias horas, tanto para Ignis como Spark.

En la Figura 4 podemos ver como las diferencias con pocos cores son casi imperceptibles en dos de los datasets, pero se vuelven más pronunciadas a medida que aumenta el nivel de paralelismo. La diferencia es menor que para el caso de Tomatula, observándose una relación entre el número de cores y el speedup no puramente lineal. En todo caso, los tiempos de Ignis son siempre menores que los obervados con Spark para todas las ejecuciones, siendo este hasta 2× más rápido que Spark para el dataset de mayor tamaño.

En el caso de PASTASpark (Figura 5), se ha utilizado como entrada uno de los ficheros incluidos en el repositorio de la herramienta (Large.Fasta, de 2 MB), así como dos del dataset de RNASim, uno de 100 secuencias de 5 MB, y otro con 1000 secuencias de 100 MB, disponible en Kim Lab Software Repository. Debemos tener en cuenta que los tiempos de ejecución mostrados aquí son solamente de la parte paralelizable (correspondiente con la etapa de alineamiento de múltiples secuencias) y que fue necesario utilizar datasets de tamaño reducido para garantizar el correcto funcionamiento de Ignis. En este caso podemos ver como Spark si supera a Ignis en rendimiento con el dataset más grande, pero no en los otros dos. Este comportamiento podría ser causado por las limitaciones debidas a los errores presentes en MPICH que afectan al rendimiento de Ignis.

Finalmente, con la aplicación Summarizer se han obtenido dos datasets de Kaggle, así como un ejemplo del propio repositorio. Fue necesario una pequeña modificación del dataset obtenido de Kaggle para ajustarlo al modelo que se esperaba en el summarizer. Esta modificación consistió simplemente en un reordenamiento de las columnas. Los datasets tienen tamaño de 10 MB, 50 MB y 100 MB. Mostramos los resultados en la Figura 6 y vemos como con determinado datasets y un número de cores pequeño, Spark supera a Ignis en rendimiento. Sin embargo, al aumentar el número de cores, el speedup vuelve a ser mayor que uno, En particular, en el caso del segundo dataset, el speedup está próximo a 3. Para Summarizer podemos ver otra vez como existe una relación más directa entre el speedup y el número de cores.

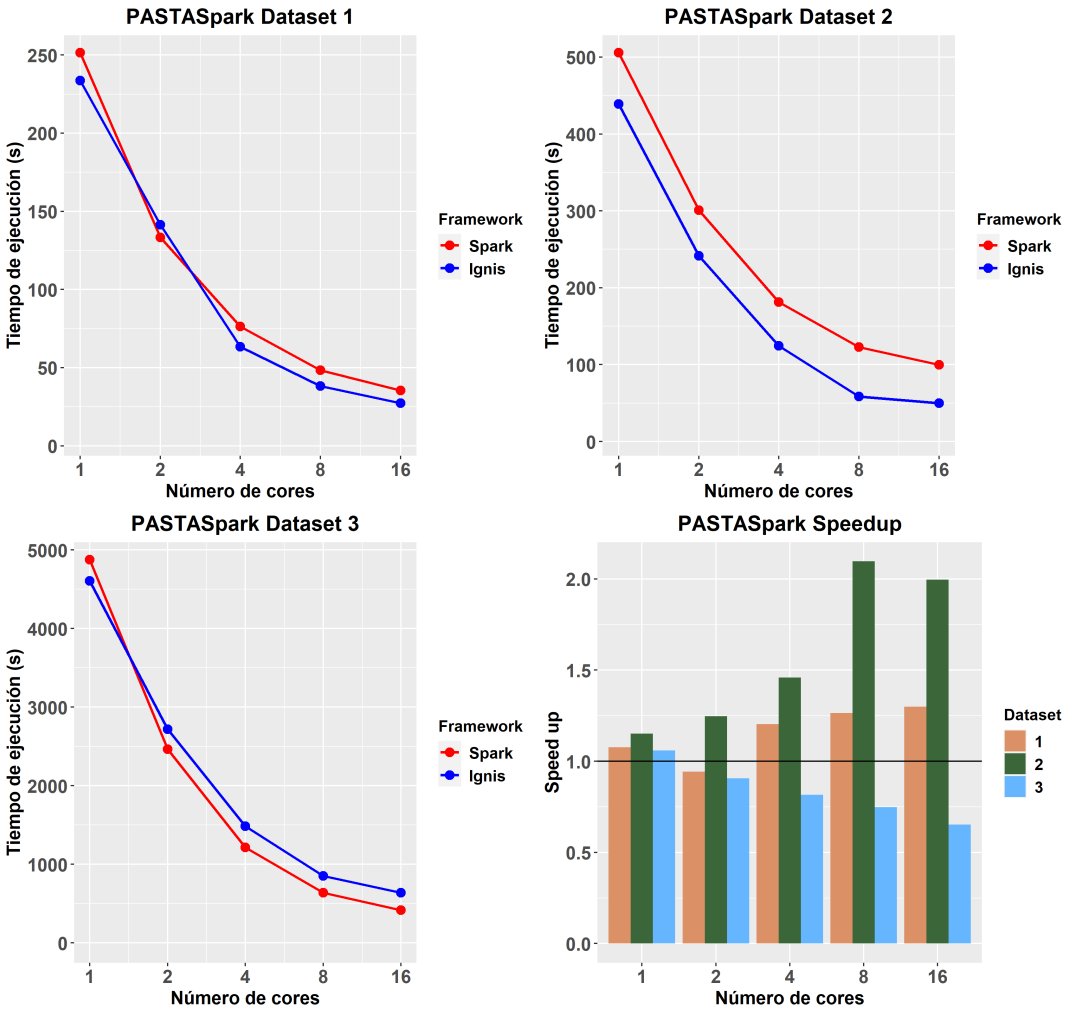

Fig. 5: Análisis de la escalabilidad de PASTASpark.

VIII. Conclusiones

Como hemos demostrado en este trabajo, Ignis es globalmente superior a Spark en cuanto a rendimiento y escalabilidad, con una programabilidad idéntica a la proporcionada por Spark. Con un esfuerzo de conversión relativamente bajo pudimos mejorar de forma significativa el rendimiento de todas las aplicaciones y, a pesar de que esta mejora varía según la aplicación y el número de ejecutores y cores asignados, es importante destacar que Ignis está todavía en fase de desarrollo y que todavía podrá ser optimizado.

Con tiempos de ejecución menores y soporte para un mayor número de lenguajes, Ignis representa una clara mejora respecto a Spark y un paso adelante en la convergencia de las áreas del Big Data y la computación de altas prestaciones (HPC). Sin embargo, también es necesario solventar algunos problemas de Ignis:

- **Eliminación de errores y bugs**. Es esencial que Ignis funcione correctamente con todos los lenguajes que soporte y todas las funciones disponibles. Es necesario entonces ir eliminando los distintos errores que vayan surgiendo en futuras pruebas así como la extensión de su API con funciones que faltan todavía del core de Spark como lookup().
- **Necesidad de soportar más lenguajes**. A pesar de que, debido a su diseño arquitectónico, Ignis podría soportar numerosos lenguajes, los módulos que soporten estos tienen que ser todavía implementados.
- **Necesidad de ofrecer herramientas de streaming, machine learning y grafos**. Spark ofrece herramientas para el desarrollo de aplicaciones con componente de streaming, machine learning y procesamiento de grafos dentro de los módulos de Spark Streaming, ML y GraphX. Para poder realmente garantizar una transición de Spark a Ignis lo menos disruptiva posible para los programadores es necesario la implementación de las funciones ofrecidas por estos paquetes.
- **Pruebas de evaluación de la escalabilidad**. Es necesario realizar más mediciones de rendimiento para las aplicaciones Summarizer y PASTASpark usando un mayor número de cores. En este trabajo se han usado hasta 16 cores, lo que representa la capacidad de un solo nodo de cómputo. Sin embargo, es necesario la ejecución usando un mayor número de nodos y cores

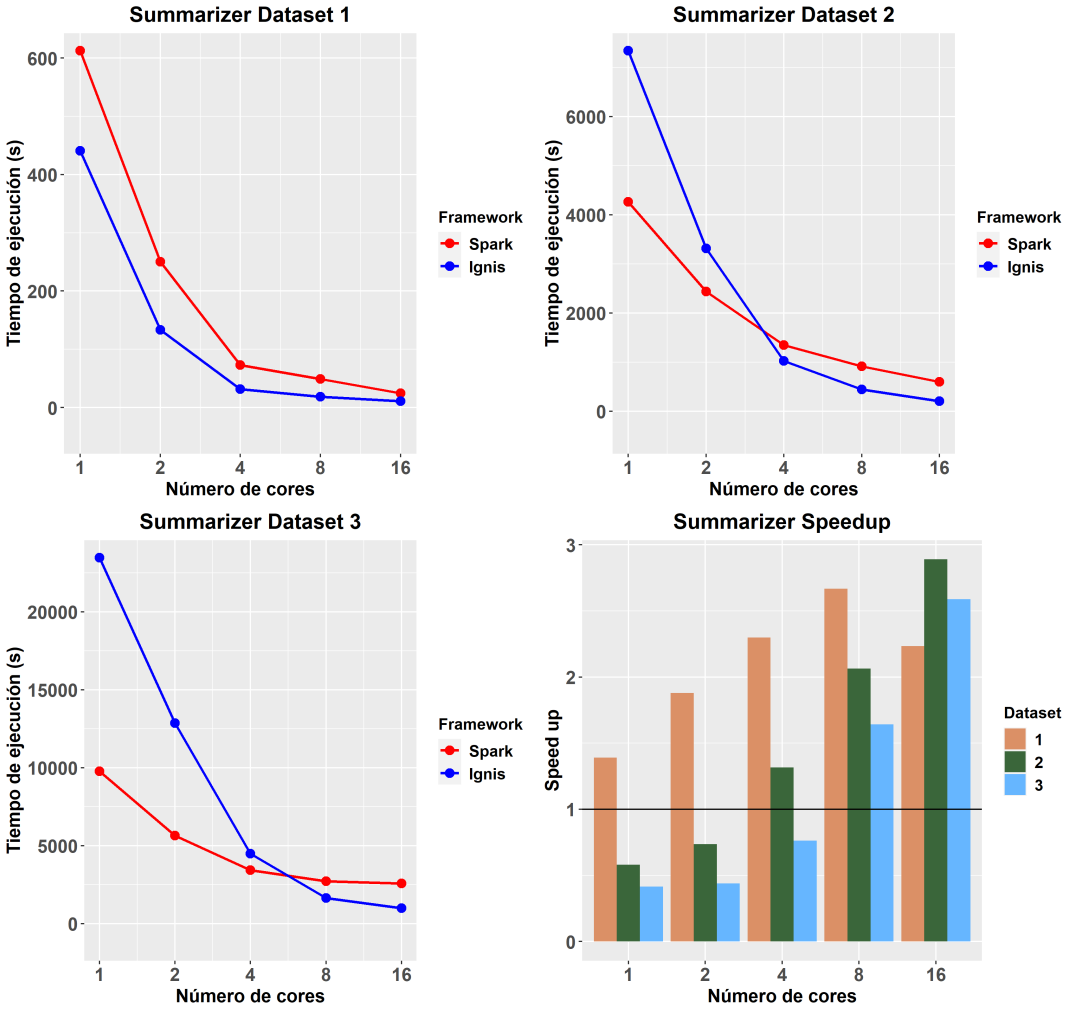

Fig. 6: Análisis de la escalabilidad de Summarizer.

para poder realizar una comparación más detallada y un análisis completo de la escalabilidad.

REFERENCES

[1] Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025, enlace, última visita: 27/07/2021

[2] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, Maggie Law, "Comparison of Next-Generation Sequencing Systems", BioMed Research International, vol. 2012, Article ID 251364, 11 pages, 2012. https://doi.org/10.1155/2012/251364

[3] Apache Hadoop, enlace, última visita: 27/07/2021

[4] Apache Flint, enlace, última visita: 27/07/2021

[5] Google BigQuery, enlace, última visita: 27/07/2021

[6] Apache Spark, enlace, última visita: 27/07/2021

[7] José Manuel Abuín, Tomás Fernández Pena, Juan Carlos Pichel, PASTASpark: multiple sequence alignment meets Big Data, Bioinformatics, Volume 33, Issue 18, 15 September 2017, Pages 2948–2950, https://doi.org/10.1093/bioinformatics/btx354

[8] José Manuel Abuín, Juan Carlos Pichel, Tomás Fernández Pena, Jorge Amigo, SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data. PLOS ONE 11(5): e0155461. (2016) https://doi.org/10.1371/journal.pone.0155461

[9] Seokjun Soe, Yoonjae Park, Heejoon Chae, BiSpark: a Spark-based highly scalable aligner for bisulfite sequencing data. BMC Bioinformatics 19, 472 (2018). https://doi.org/10.1186/s12859-018-2498-2

[10] Andrian Yang, Michael Troup, Peijie Lin, Joshua W K Ho, Falco: a quick and flexible single-cell RNA-seq processing framework on the cloud, Bioinformatics, Volume 33, Issue 5, 1 March 2017, Pages 767–769, https://doi.org/10.1093/bioinformatics/btw732

[11] Java, enlace, última visita: 27/07/2021

[12] Scala, enlace, última visita: 27/07/2021

[13] Python, enlace, última visita: 27/07/2021

[14] R, enlace, última visita: 27/07/2021

[15] César Piñeiro, Rodrigo Martínez-Castaño, Juan Carlos Pichel, Ignis: An efficient and scalable multi-language Big Data framework, Future Generation Computer Systems, Volume 105, 2020, Pages 705-716, ISSN 0167-739X, https://doi.org/10.1016/j.future.2019.12.052.

[16] Fortran, enlace, última visita: 27/07/2021

[17] Apache Thrift, enlace, última visita: 27/07/2021

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. DOI:https://doi.org/10.1145/1327452.1327492

[19] HDFS, enlace, última visita: 27/07/2021

[20] YARN, enlace, última visita: 27/07/2021

[21] Docker, enlace, última visita: 27/07/2021

[22] Go, enlace, última visita: 27/07/2021

[23] Apache Spark SQL, enlace, última visita: 27/07/2021

[24] Apache Spark Streaming, enlace, última visita: 27/07/2021

[25] Apache Spark MLlib, enlace, última visita: 27/07/2021

[26] Apache Spark GraphX, enlace, última visita: 27/07/2021

[27] Aikaterini Boufea, Richard Finkers, Martijn van Kaauwen, Mark Kramer and Ioannis N. Athanasiadis, Managing Variant Calling Files the Big Data Way: Using HDFS and Apache Parquet, Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing,

Applications and Technologies (BDCAT '17). ACM, 2017, p. 219-226, doi:10.1145/3148055.3148060
[28] Github Tomatula, enlace, última visita: 27/07/2021
[29] Lladós Segura, Jordi; Guirado Fernández, Fernando; Cores Prado, Fernando. (2017) . PPCAS: Implementation of a Probabilistic Pairwise Model for Consistency-Based Multiple Alignment in Apache Spark. Lecture Notes in Computer Science, 2017, vol. 10393, p. 601–610. https://doi.org/10.1007/978-3-319-65482-9-45.
[30] Github PPCAS, enlace, última visita: 27/07/2021
[31] CTypes, enlace, última visita: 27/07/2021
[32] Github PASTASpark, enlace, última visita: 27/07/2021
[33] Yihong Gong, Xin Liu: Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis. Proceedings of the 24 th annual international ACM SIGIR conference on Research and development in information retrieval, New Orleans, Louisiana, United States 2001, pp. 19-25
[34] Github Summarizer, enlace, última visita: 27/07/2021
[35] 2to3, enlace, última visita: 27/07/2021 https://www.overleaf.com/project/60e1cfb31835a3281317e6ee
[36] DendroPy, enlace, última visita: 27/07/2021
[37] MPICH, enlace, última visita: 27/07/2021
[38] HmmerAlign, enlace, última visita: 27/07/2021
[39] MAFFT, enlace, última visita: 27/07/2021
[40] NCBI, enlace, última visita: 27/07/2021
[41] 1000Genomes, enlace, última visita: 27/07/2021
[42] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, Minyi Guo, More convenient more overhead: The performance evaluation of Hadoop streaming, in: Proc. of the ACM Symposium on Research in Applied Computation, 2011, pp. 307–313.