UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

Departamento

Inteligencia Artificial

TRABAJO DE DIPLOMA

Título del trabajo Aplicación del modelo de programación Spark al cálculo de medidas de similitud para pares de genes

Autor del trabajo Alejandro Arteaga Pérez

Tutores del trabajo Dra. Deborah Galpert Cañizares

Dr. Reinaldo Molina Ruiz

20/06/2018

Santa Clara

Resumen

Las técnicas de Minería de Datos han sido adaptadas para manejar grandes volúmenes de datos mediante modelos de programación como MapReduce y Spark. La extracción de datos a partir de secuencias de proteínas en la genómica comparativa es uno de los procesos dentro de la minería de datos que resulta imprescindible en la Bioinformática. En este trabajo se utilizó Spark para abordar el problema de la comparación de pares de proteínas, en específico, para calcular descriptores de proteínas. Esta elección se debe fundamentalmente a que Spark puede reducir el tiempo de ejecución en el problema planteado al soportar varios tipos de trabajos computacionales, procesamiento de flujo de datos, manejo intensivo de memoria y una conexión a un clúster de Hadoop para manejar los datos distribuidos en el sistema de archivos HDFS. Algunos experimentos de cálculo de descriptores en proteomas de levaduras fueron realizados teniendo en cuenta la configuración del clúster de la Universidad Central “Marta Abreu” de Las Villas. Los resultados obtenidos en cuanto a los tiempos de ejecución son prometedores para poder aplicar el programa de cálculo de descriptores a múltiples proteomas.

Palabras Clave: Spark, clúster, descriptores de proteínas.

Abstract

Data mining techniques have been adapted to handle large volumes of data through programming models such as MapReduce and Spark. Data extraction from protein sequences in comparative genomics is an essential data mining process required in Bioinformatics. Spark was used in this work to tackle the pairwise protein comparison problem, specifically, the protein descriptor calculations. The reason of this selection is related with the fact that Spark may reduce the execution time in the proposed problem by supporting different types of computational tasks, data flow process, intensive memory usage and connections to a Hadoop cluster in order to manage the data distributed in the HDFS file system. Some experiments calculating protein descriptors in yeast proteomes were executed considering the settings of the computing cluster of the Universidad Central “Marta Abreu” de Las Villas. The results obtained regarding execution times are promising to carry out further calculations of protein descriptors in multiple proteomes.

Keywords: Spark, cluster, protein descriptors.

Antes pensábamos que nuestro futuro estaba en las estrellas. Ahora sabemos que está en nuestros genes.

James Watson

¡Hemos encontrado el secreto de la Vida!

En la mañana del 28 de Febrero de 1953, Francis Crick entra en el pub *The Eagle* de Cambridge exclamando, preso de excitación, esta famosa frase. Ese día, que algunos han llamado "el octavo día de la creación", resolvió finalmente, junto con James Watson, la estructura tridimensional del DNA.

Agradecimientos

A mis tutores la Dra. Deborah Galpert Cañizares y el Dr. Reinaldo Molina Ruiz y a todos los que a través de estos arduos años de estudio me han ayudado y apoyado siempre que los he necesitado. ¡Muchas gracias!

Dedicatoria

Dedicado a todas las personas que han hecho posible que me gradúe de Ciencia de la Computación, en especial a mi familia.

Índice

Índice de Figuras

Índice de Tablas

Introducción

Las tecnologías de big data están siendo aplicadas en diferentes ramas de la Bioinformática desde la perspectiva de Big Data Analytics (Kashyap et al., 2014). El análisis de datos de múltiples genomas secuenciados es uno de los retos en la Bioinformática para realizar estudios evolutivos. En específico, dentro de la genómica comparativa se realizan comparaciones entre genomas para encontrar regiones conservadas que conllevan al manejo de grandes volúmenes de datos por lo que se realizan esfuerzos por implementar versiones big data del alineamiento de secuencias (Matsunaga et al., 2008), información básica que se ha tomado en cuenta en la comparación de secuencias.

En la comparación de genomas se tratan de detectar genes llamados ortólogos, que han evolucionado a partir de un ancestro común y permiten estudiar la función de proteínas desconocidas. El reto de encontrar estos genes entre múltiples genomas o proteomas a partir de comparaciones par a par entre los genes o proteínas de pares de genomas o proteomas ha sido tratado por la comunicad científica como un problema cuadrático en (Sonnhammer et al., 2014, Kristensen et al., 2011) y ha sido abordado en diferentes implementaciones para clúster. Se han publicado algoritmos para la detección de ortólogos desde la perspectiva de big data como los mencionados en (Wall et al., 2010).

En el grupo de bioinformática de la Universidad Central “Marta Abreu” de las Villas (UCLV) se han realizado investigaciones con herramientas de aprendizaje automatizado para el análisis de datos de proteínas en la detección de ortólogos con la colaboración del Grupo de Inteligencia Artificial de la Universidad de Granada (Galpert, 2016), (Galpert et al., 2017), (Galpert et al., 2018). Además se realizó el Trabajo de Diploma (Zunda-Herrera and Galpert, 2016) como un primer intento de utilizar el clúster de la UCLV, específicamente, una versión experimental de un clúster de Spark.

El mencionado análisis de datos de proteínas en la UCLV ha conllevado al cálculo de descriptores o medidas de similitud entre pares de proteínas teniendo en cuenta su naturaleza cuadrática. En (Millo and Galpert, 2012) se implementó el cálculo en paralelo y se estudia la escalabilidad de estos cálculos. Luego en (Landaburo-Del-Arco et al., 2016) se amplía el estudio de la escalabilidad a varios pares de proteomas y se muestra un diseño utilizando MapReduce como modelo de big data, a su vez, se

estudia la posibilidad de la aplicación de este diseño en el contexto del clúster de la UCLV en el momento de realización de dicho Trabajo de Diploma. Por otra parte, en la propia Universidad se han implementado programas para el cálculo de diferentes medidas de similitud libres de alineamiento entre pares de proteínas (Molina et al., 2011). Este software TI2BioP del Centro de Bioactivos Químicos cuenta con una versión paralela que requiere computadoras con altas prestaciones para su ejecución sin aprovechar aún los recursos disponibles en el clúster. Dado el nuevo contexto de dicho clúster, que en el momento actual cuenta con un clúster de Spark como tecnología big data y una configuración más avanzada aparece el siguiente problema de investigación.

Problema de investigación

¿Cómo implementar los diversos cálculos de descriptores de proteínas teniendo en cuenta la configuración del clúster de Spark de la Universidad Central “Marta Abreu” de Las Villas?

Objetivo general

Aplicar el modelo de programación Spark al cálculo de descriptores de proteínas considerando la configuración del clúster de computadoras de la Universidad Central “Marta Abreu” de Las Villas.

Objetivos específicos

1. Estudiar el modelo de paralelización propuesto en trabajos anteriores.
2. Identificar las ventajas del modelo Spark para la solución del problema propuesto.
3. Seleccionar la infraestructura adecuada para la ejecución del programa distribuido.
4. Diseñar las funciones del cálculo de descriptores de proteínas.
5. Implementar las funciones del cálculo de descriptores utilizando el modelo Spark.
6. Realizar pruebas a las funciones implementadas.

Tareas de investigación

1. Revisión de las implementaciones de cálculo de medidas de similitud existentes en la UCLV.
2. Estudio del modelo Spark con relación a otras alternativas.
3. Estudio de las características del clúster en la UCLV.

4. Diseño e implementación de los cálculos de descriptores de proteínas utilizando tecnología de big data.
5. Probar los cálculos en el clúster de Spark.

Aporte práctico

Las medidas a implementar en este trabajo pueden contribuir a agilizar y hacer escalables los análisis de datos de proteínas permitiendo el procesamiento de grandes volúmenes de secuencias de múltiples proteomas.

Aporte metodológico

La metodología aplicada para la implementación de los cálculos en un clúster de Spark puede servir de base para futuras implementaciones, así como para otras investigaciones incluso fuera del campo de la Bioinformática, aprovechando los recursos de cómputo de la UCLV.

Estructura de la tesis

Este Trabajo de Diploma ha sido estructurado en tres capítulos. El Capítulo I "Big Data para el cálculo de medidas de similitud entre pares de proteínas en la comparación de proteomas" aborda los aspectos teóricos esenciales para la implementación de estos cálculos utilizando la tecnología big data, en específico, el modelo Spark. En el Capítulo II "Implementación Spark del cálculo de descriptores de proteínas" se presentan las transformaciones realizadas a la versión paralela del software TI2BioP para su implementación en Spark. Finalmente, en el Capítulo III "Experimentación" se muestran los resultados experimentales de la ejecución de la nueva implementación Spark.

Capítulo 1. Big Data para el cálculo de medidas de similitud entre pares de proteínas en la comparación de proteomas

En este capítulo inicialmente se tratan aspectos relacionados con la complejidad computacional del cálculo de medidas de similitud para comparar pares de genes dentro de la genómica comparativa. Seguidamente, se presentan las características de la implementación paralela realizada en la UCLV con sus limitaciones en cuanto al modelo de programación empleado para su desarrollo. Finalmente, se especifican las características de Spark como modelo de programación para big data elegible para lograr una implementación escalable del cálculo de dichas medidas.

1.1 Complejidad computacional y evolución de las implementaciones de la comparación de secuencias de proteínas

La comparación de secuencias de proteínas se presenta como un problema de la Bioinformática dentro de la genómica/proteómica comparativa. Requiere de la construcción y el uso de herramientas de minería de datos para el análisis de especies relacionadas con vistas a identificar elementos funcionales, como los genes ortólogos que codifican proteínas, “en virtud de” su fuerte conservación a través de la evolución (Kamvysselis, 2003). Esta clasificación tiene el fin de descubrir las relaciones evolutivas entre los genomas y las funciones de proteínas aún desconocidas. Las comparaciones de pares de proteínas muestran una escalabilidad cuadrática que debe ser abordada desde la programación paralela y los modelos de programación para big data (Kristensen et al., 2011, Sonnhammer et al., 2014).

La clasificación de genes ortólogos requiere del manejo de grandes volúmenes de datos de manera escalable. En este tipo de clasificación se recibe como datos de entrada proteomas enteros con miles de proteínas y se debe realizar minería de estos datos para encontrar pares o grupos de estos genes que además son muy escasos. La clasificación parte del procesamiento de las secuencias de aminoácidos de cada proteoma a comparar con vistas a representar las proteínas en forma de vectores numéricos. Luego estos vectores pueden ser utilizados en la comparación de pares de proteínas mediante medidas de similitud basadas en el libre alineamiento de las mismas. Otros tipos de comparaciones de secuencias se basan en el alineamiento de secuencias de aminoácidos sin convertirlas a vectores numéricos (Galpert et al.,

2015). Las primeras han mostrado varias ventajas sobre las basadas en alineamiento como: no son sensibles a los reordenamientos de los genomas, (ii) permiten detectar señales funcionales cuando las secuencias tienen baja similitud y (iii) generalmente son menos complejas computacionalmente y menos consumidoras de tiempo (Mahmood et al., 2012, Vinga, 2003). Es por esto que el presente trabajo está enfocado fundamentalmente en estas medidas, aunque la propuesta de implementación big data que se realiza puede ser generalizada al cálculo de medidas basadas en alineamiento en la comparación de pares de proteínas.

En un análisis de la complejidad computacional de las comparaciones par a par de proteínas con medidas de similitud basadas en alineamiento (implementado con programación dinámica) realizado en (Goya et al., 2016) se ha estimado un orden de $O(\mathrm{N} \times \mathrm{m} \times \mathrm{n})$ donde N es el total de pares de secuencias a comparar y m y n las máximas longitudes de secuencias de los dos proteomas en comparación. Por otra parte, la paralelización de los mismos reduce este orden a $O(\frac{\mathrm{N} \times \mathrm{m} \times \mathrm{n}}{\mathrm{p}} + \mathrm{N} \times (\mathrm{n} + \mathrm{m}))$ donde p es la cantidad de procesadores y el cálculo de cada par se distribuye en cada procesador de manera independiente. En otra variante de medida basada en alineamiento, donde se mide la similitud de la energía de contacto de aminoácidos en regiones sin gaps entre dos secuencias alineadas, se ha estimado el costo para su versión secuencial en el orden de $O(\mathrm{N} \times (\mathrm{m} + \mathrm{n})^2)$ y una reducción del mismo a $O(\frac{\mathrm{N} \times (\mathrm{m}+\mathrm{n})^2}{\mathrm{p}} + \mathrm{N} \times (\mathrm{n} + \mathrm{m}))$ mediante el mismo tipo de paralelización.

Teniendo en cuenta el análisis de escalabilidad realizado al cálculo de las medias basadas en alineamiento en (Landaburo-Del-Arco et al., 2016), la habilidad del programa paralelo en ejecución de mantener la eficiencia en un valor constante al incrementar simultáneamente el número de procesadores y el tamaño del problema se logra en buena medida en la estimación realizada con el esquema de paralelización comentado anteriormente. De ahí que este esquema pueda considerarse escalable horizontalmente y dicha escalabilidad pudiera hacerse efectiva si se desarrolla una implementación para ejecutar en clúster con técnicas de manejo de big data.

En cuanto a las medidas libres de alineamiento, si se maneja una secuencia pesada X de longitud n y con un patrón s de longitud m entonces la ocurrencia de s en X puede ser localizada en $O((\mathrm{n} + \mathrm{m}) \times \log \mathrm{n})$ de tiempo y espacio lineal. En este caso, una secuencia X pesada de longitud n consiste en un conjunto de pares $(c, \pi_i(c))$ siendo $\pi_i(c)$ la probabilidad de que aparezca el carácter c en la posición i, con 1 ≤ i ≤ n y $\sum_i^n \pi_i(c) = 1$. Este peso $\pi_i(c)$ también puede ser modelado como la estabilidad de la

contribución del carácter al complejo molecular (Elloumi and Zomaya, 2011). De este modo, la complejidad del cálculo de descriptores libres de alineamiento será $O(\mathrm{r} \times (\mathrm{n1} + \mathrm{n2}) \times (\mathrm{n} + \mathrm{l}) \times \log \mathrm{n})$, donde n es la máxima longitud de secuencias de los dos proteomas con $\mathrm{n1}$ y $\mathrm{n2}$ secuencias, respectivamente, l longitud máxima de patrones a buscar en las secuencias, r total de patrones a buscar. Si se quiere realizar la comparación par a par de proteínas utilizando los descriptores libres de alineamiento, entonces la complejidad computacional es $O(\mathrm{N} \times \mathrm{d})$, donde d es el total de descriptores a utilizar en la comparación. A pesar de que el orden de complejidad es menor para estas medidas libres de alineamiento la escalabilidad de la implementación depende igualmente del modelo de programación elegido y de la infraestructura empleada en la ejecución de los cálculos.

En (Cattaneo et al., 2015) los autores se refieren a la necesidad del uso de big data tanto en las comparaciones de secuencias basadas en alineamiento como en las libres de alineamiento. Mencionan implementaciones realizadas usando la computación de alto rendimiento (High Performance Computing o HPC en inglés) como un sistema de procesamiento en paralelo o distribuido que consiste en un conjunto de ordenadores interconectados trabajando juntos cooperativamente como un único recurso informático integrado. El objetivo general de HPC es la ejecución aplicaciones con mayor rapidez o la ejecución de los problemas que no se pueden ejecutar en un único servidor. Con el uso computación para nube (Bonvin, 2012) se cuenta con múltiples nodos que cumplen diversas funciones en el clúster y que están conectados por algún tipo de red de alta velocidad. Uno de los ejemplos de software de comparación de secuencias implementado para nube de computadora es el conocido BLAST (Matsunaga et al., 2008) que desarrolla un algoritmo heurístico para el alineamiento de secuencias devolviendo un subconjunto de secuencias de alta similitud con una secuencia de búsqueda o para realizar comparaciones de pares de secuencias igualmente seleccionando subconjuntos de mejores resultados en la comparación. Esta versión de BLAST utiliza el modelo MapReduce para big data (Dean et al., 2004) como alternativa a los modelos de paralelización para HPC como (Message Passing Interface MPI en inglés) (Snir and Otto, 1998) o (Compute Unified Device Architecture CUDA en inglés (Nickolls et al., 2008).

Según (Fernández et al., 2014) las soluciones de HPC basadas en modelos de programación paralela como MPI y CUDA presentan dificultades en el acceso a los datos y en la facilidad de desarrollo del software por los requerimientos y restricciones de los esquemas de programación disponibles. En particular, para el cálculo de medidas de similitud entre proteínas libres de alineamiento como el cálculo similitud de

la frecuencia de sub-cadenas de longitud k, es decir, k-mers (k-words) entre un par o un grupo de secuencias se requiere calcular progresivamente la cantidad de k-mers encontrados en las secuencias de todas las posibles permutaciones de aminoácidos en un patrón de k caracteres. Con este propósito, determinadas informaciones temporales deben ser cargadas a la memoria principal (Cattaneo et al., 2015). Esto impondría una barrera en el cálculo de la similitud entre todos los pares de proteínas de dos proteomas ya que los datos usualmente están almacenados en diferentes localizaciones y se necesitaría una comunicación en red intensiva para accederlos, así como otros costos de entrada/salida. Aun cuando se logre superar esta barrera, se necesitaría una gran cantidad de memoria para almacenar los datos pre-cargados para el cómputo. Se necesitaría entonces un mecanismo más robusto de tolerancia a fallas que el de MPI o el de CUDA, modelos caracterizados por múltiples accesos a disco y limitados en cuanto a la escalabilidad (Fernández et al., 2014). Para tratar dichas dificultades, han emergido soluciones de programación paralela para el manejo de datos a gran escala como el modelo MapReduce y más recientemente el modelo Apache Spark (Zaharia et al., 2012). Este último es un marco de trabajo en memoria para el procesamiento paralelo de datos en aplicaciones iterativas e interactivas que pudiera ser utilizado en el cálculo de medidas de similitud entre proteínas en la UCLV por las ventajas que ofrece y por la disponibilidad de un clúster de Spark en dicho centro.

1.2 Software de comparación de secuencias de proteínas en la UCLV

Algunas medidas de similitud basadas en el libre alineamiento de secuencias aparecen implementadas por parte del grupo de bioinformática de la UCLV en https://sourceforge.net/projects/ab-af-ortholog-detection/. Estas medidas se listan a continuación, Las denotadas (5-10) se encuentran definidas en PROFEAT-Protein Feature Server (Rao et al., 2011) mientras que las basadas en los mapas de cuatro colores y los descriptores Nandy (1-2) pueden ser calculados mediante un programa de codificación gráfico-numérica (Molina et al., 2011) disponible en https://sourceforge.net/projects/ti2biop/.

1. Descriptores de mapa de cuatro colores: descriptores topológicos (series de momentos espectrales) derivados de mapas de cuatro colores para proteínas (Agüero-Chapin et al., 2013 Jul 16).

2. Descriptores Nandy: descriptores topológicos (series de momentos espectrales) derivados de mapas Cartesianos de proteínas (Agüero-Chapin et al., 2011 Feb 1).
3. k-mers o k-words: frecuencia de cada subsecuencia o palabra de longitud fija k en un conjunto de secuencias (Gunasinghe et al., 2014).
4. k-mers espaciados o palabras espaciadas: k-mers contiguos con "caracteres no importantes" en posiciones fijas o predeterminadas en un conjunto de secuencias (Leimeister et al., 2014).
5. Composición de aminoácidos: fracción de cada aminoácido en una proteína (Bhasin and Raghava, 2004, Kumar et al., 2008).
6. Pseudo composición de aminoácidos de Chou: es una mejora del descriptor de composición de aminoácidos al añadir información sobre el orden de la secuencia al vector de frecuencia de aminoácidos (Chou, 2001). El parámetro λ es la distancia topológica entre dos aminoácidos dentro de la secuencia considerando el concepto de la pseudo composición de aminoácidos donde el efecto del orden de la secuencia se integra con la composición de aminoácidos, λ < longitud de la proteína.
7. Auto correlación de Geary**:** Auto correlación cuadrada de propiedades de aminoácidos a través de la secuencia (Sokal and Thomson, 2006 Jan)**.**
8. Auto correlación de Moran: Auto correlación de las propiedades de los aminoácidos en la secuencia (Horne, 1988 Mar).
9. Auto correlación total: descriptores de Auto correlación (Geary, Moran y Moreau-Broto) basados en determinadas propiedades de los aminoácidos que son normalizadas en conjunto (Cao et al., 2013).
10. Descriptores de composición (C), Transición (T) y Distribución (D) (CTD): información de la división de aminoácidos en tres clases de acuerdo al valor de sus atributos como la hidrofobicidad, el volumen normalizado de van der Waals, la polaridad, etc. Cada aminoácido se clasifica por cada uno de los índices en las clases 1, 2 y 3. En los descriptores C se mide el porciento global de cada clase codificada en la secuencia, en los T se mide el porcentaje de frecuencia en que la clase 1 es seguida por la clase 2 o la clase 2 es seguida de la 1 en la secuencia codificada. En los descriptores D se mide la distribución de cada atributo en la secuencia codificada (Dubchak et al., 1995, Dubchak et al., 1999).
11. Descriptores Quasi-Sequence-Order (QSO): combinación de composición de secuencia y correlación entre las propiedades de los aminoácidos definido por Chou KC (2000) (Chou, 2000).

Las implementaciones de estos descriptores están realizadas en Phyton utilizando la biblioteca de paralelización basada en CUDA. Este modelo de programación presenta buenas prestaciones en cuanto a que permite lecturas dispersas, es decir, permite consultar cualquier posición de memoria, presenta ancho de banda de memoria local muy rápido y funciona con la CPU como coprocesador. Sin embargo, muestra limitaciones relacionadas con: que no se puede utilizar recursividad, en precisión simple no soporta números desnormalizados o NaNs, puede existir un "cuello de botella" entre la CPU y la GPU por los anchos de banda de los buses y sus latencias, los hilos de ejecución, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total y los núcleos típicamente corren el mismo código (Nickolls et al., 2008). Por estas limitaciones recogidas en la literatura así como por el tiempo real constatado en los cálculos para comparaciones de millones de pares de proteínas con una única máquina es que se ha valorado utilizar el modelo Spark en un clúster de computadoras.

1.3 Spark como modelo de programación para big data

Spark extiende el modelo MapReduce soportando eficientemente más tipos de trabajos computacionales, incluyendo consultas interactivas y procesamiento de flujo de datos. La velocidad es importante cuando se trata de procesar grandes conjuntos de datos y es una de las principales características que Spark ofrece, para esto hace un uso intensivo de la memoria computacional (Karau et al., 2015).

Por su parte, el modelo MapReduce organiza el procesamiento mediante las funciones Map y Reduce definidas en términos de pares (llave, valor) donde función Map toma un par (llave, valor) para cada elemento en la entrada y genera como salida un conjunto de pares intermedios (llave, valor). Luego el marco de trabajo MapReduce recolecta todos los pares intermedios y los agrupa de acuerdo a la llave, generando un grupo para cada llave que constituye la entrada para la función Reduce. Esta última se aplica en paralelo a cada grupo producido por la función Map, toma estos grupos y los combina para generar los correspondientes pares (llave, valor) como salida final del procedimiento MapReduce. De este modo, el programador se limita al desarrollo de las funciones Map y Reduce dejando todos los detalles de comunicación, balance de carga, asignación de recursos, inicio de trabajo y distribución de archivos a la plataforma subyacente. MapReduce puede ejecutarse en un clúster formado por una arquitectura maestro/esclavo, que posea un sistema de archivos distribuidos, como (Hadoop Distributed File System por sus siglas en inglés) (HDFS). Sin embargo, no todos los algoritmos pueden ser expresados de acuerdo al esquema de MapReduce,

por ejemplo, los algoritmos iterativos (Lin, 2013). Además, la entrada a cualquier tarea MapReduce vinculada con Hadoop, se almacena en HDFS, por tanto, cada vez que se ejecuta una tarea, la entrada debe ser recargada de disco. Esto se realiza independientemente de si esta entrada ha cambiado o no en las iteraciones anteriores, lo que provoca una penalidad en el funcionamiento (Fernández et al., 2014).

En cambio, la principal abstracción de programación en Spark son los llamados Resilient Distributed Dataset (RDD), colección de objetos inmutables y distribuidos. Cada RDD es particionado en varias nuevas particiones, las cuales son procesadas en diferentes nodos del clúster. Los RDDs pueden contener varios tipos de objetos, que pueden ser de distintos lenguajes, ya sea Java, Scala, o Python, además de clases definidas por el usuario. En Spark el procesamiento se expresa creando nuevos RDDs, transformando RDDs existentes, o llamando operaciones sobre un RDD para obtener un resultado. Oculto a la vista del usuario, Spark automáticamente distribuye los datos contenidos en los RDDs en el clúster y paraleliza las operaciones que realiza sobre los mismos. A su vez, Spark puede ejecutar sobre un clúster Hadoop y acceder a cualquier fuente de datos Hadoop (Karau et al., 2015). Spark reutiliza datos a través de múltiples cómputos, a la vez que mantiene la escalabilidad y tolerancia a fallas de MapReduce (Fernández et al., 2014).

En Spark se pueden crear RDDs de dos formas, una es cargando datos externos, y otra es distribuyendo una colección de objetos en el programa controlador (llamado driver). Una vez creados, los RDDs ofrecen dos tipos de operaciones sobre ellos, transformaciones y acciones, las primeras construyen un nuevo RDD a partir de uno ya existente, y las segundas computan resultados basados en un RDD, y estos son devueltos al programa controlador o salvados en un sistema de archivos externo (Karau et al., 2015). La Tabla 1. 1 contiene las transformaciones básicas sobre RDD mientras que la Tabla 1. 2 contiene las acciones básicas a realizar sobre un RDD.

Tabla 1. 1 Transformaciones básicas sobre RDD.

Función	**Descripción**
`Map`	Aplica una función a cada elemento del RDD y retorna un nuevo RDD con el resultado.
`Flatmap`	Aplica una función a cada elemento del RDD y retorna un RDD con los contenidos de los iteradores retornados. Son utilizadas para extraer palabras.
`Filter`	Retorna un RDD que contiene los elementos que

	pasaron la condición enviada para filtrar.
`Distinct`	Remueve duplicados.
`sample(withReplacement, fraction, [seed])`	Toma una muestra de un RDD.
`Unión`	Produce un RDD que contiene los elementos de dos RDD.
`Intersection`	Retorna un RDD que contiene sólo los elementos que coinciden en dos RDD.
`Subtract`	Remueve los elementos contenidos en un RDD. Por ejemplo: datos de entrenamiento.
`Cartesian`	Producto cartesiano con otro RDD.

Tabla 1. 2 Acciones básicas sobre RDD.

Función	**Descripción**
`collect()`	Retorna todos los elementos de un RDD.
`count()`	Retorna el número de elementos en un RDD.
`take(num)`	Retorna tantos elementos del RDD como sea el valor de num.
`top(num)`	Retorna los top num elementos del RDD.
`takeOrdered(num)(ordering)`	Retorna num elementos ordenados según ordering.
`takeSample(withReplacement, num, [seed])`	Retorna num elementos aleatorios.
`reduce(func)`	Combina los elementos del RDD en paralelo.
`fold(zero)(func)`	Igual que reduce pero con el zero value indicado.
`aggregate(zeroValue)(seqOp, combOp)`	Similar a reduce pero utilizado para retornar un tipo diferente.

Las transformaciones y las acciones son diferentes a causa de la forma en que Spark maneja computacionalmente los RDDs, aunque se puedan definir nuevos RDDs en cualquier momento, Spark solo procesa los mismos en modo perezoso, computándose estos la primera vez que son utilizados en una acción (Karau et al., 2015). Esta característica es muy útil cuando se trabaja con enormes cantidades de datos como es el caso del manejo de big data.

Una de los atributos más importantes de Spark es que puede “cachear” datos en memoria por todo el clúster, esta es una de las más importantes utilidades de los RDDs. Cacheando los RDDs Spark los mantiene en memoria, así la primera vez que éstos son utilizados en una acción, son cargados de la fuente de los datos, y cuando es necesaria la reutilización de los mismos, los datos se acceden directamente desde la memoria, evitando así las altas penalizaciones de entrada/salida. Esta propiedad es muy importante cuando se realiza un trabajo sobre grandes conjuntos de datos, o se utilizan algoritmos de Aprendizaje Automatizado que requieren varias pasadas sobre los mismos datos (Pentreath, 2015).

Conjuntamente con los RDDs, Spark utiliza otra abstracción computacional, las variables compartidas, que pueden ser utilizadas en operaciones paralelas. Cuando Spark ejecuta alguna función en paralelo como conjunto de tareas en diferentes nodos, carga una copia de cada variable utilizada en la función para cada tarea, y en algunas ocasiones alguna de estas variables necesita ser compartida entre tareas, o entre las tareas y el programa controlador. Los dos tipos de variables compartidas que soporta Spark son: las broadcast, las cuales pueden ser utilizadas de caché a valor en memoria en todos los nodos, y los acumuladores, las cuales son variables a las que sólo se les puede incrementar como contadores o sumadores (Espinosa, 2015).

Spark en modo distribuido utiliza una arquitectura maestro/esclavo con un controlador central y varios nodos llamados trabajadores (workers). El controlador es el encargado de comunicarse con los nodos trabajadores distribuidos, llamados ejecutores (executors). El controlador ejecuta sobre su propio proceso de Java, mientras que los nodos ejecutores ejecutan en procesos Java independientes. En conjunto, el controlador y los ejecutores conforman una aplicación de Spark. Las aplicaciones Spark son lanzadas en un conjunto de máquinas usando un servicio externo llamado cluster manager. El controlador es el proceso desde el cual se ejecuta el main() de una aplicación, además, es el proceso que maneja el código del usuario encargado de crear el SparkContext, objeto que permite al controlador la conexión con Spark. El SparkContext básicamente representa una conexión con un clúster de computadoras

que usa Spark. El controlador, a su vez, maneja el código del usuario que crea las RDDs y realiza transformaciones y acciones, así como es el encargado de terminar la corrida de la aplicación y liberar los recursos del clúster (Karau et al., 2015).

En particular en Python existe la biblioteca PySpark, herramienta lanzada por Apache Spark Community, que se conecta a JVM Spark usando una mezcla de tuberías en los trabajadores y Py4J. Esta última es una biblioteca especializada para la interoperabilidad Python/Java en el controlador. Esta arquitectura oculta un gran número de complejidades involucradas en hacer a PySpark funcionar, como muestra la Figura 1. 1 . Uno de los mayores retos es que una vez que la información ha sido copiada del trabajador de Python para la JVM, no está en una forma que la JVM fácilmente pueda parsear. Esto requiere un manejo especial tanto del trabajador de Python como de Java para asegurar información suficiente como para que la partición esté disponible en la JVM. PySpark ofrece el PySpark Shell que vincula la API de Python con el núcleo de Spark e inicializa el contexto de Spark (Karau and Warren, 2017).

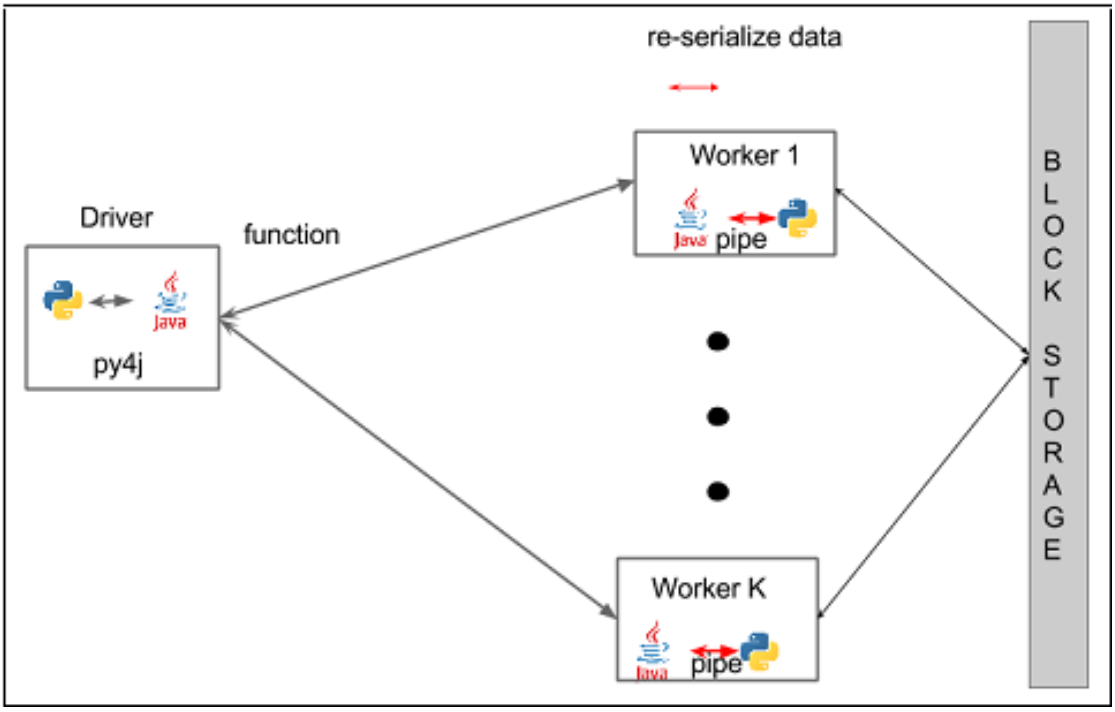

Figura 1. 1 Arquitectura PySpark. Tomado de (Karau et al., 2015).

1.4 Consideraciones finales del Capítulo

En este capítulo se presentan los aspectos teóricos básicos para implementar los cálculos de medidas de similitud entre pares de genes mediante la tecnología big data.

Primeramente, se presenta un resumen de la complejidad computacional de estos cálculos reportada en la literatura, lo que justifica la utilización de big data para el problema planteado de superar la implementación actual realizada con programación paralela. Se describieron las principales características de los modelos de programación para big data como MapReduce y Spark, así como su interacción con el sistema de archivos distribuidos HDFS. Se especifican las ventajas de Spark ejecutándose en un clúster para el trabajo con grandes volúmenes de datos por las facilidades de manejo intensivo de memoria y la velocidad de procesamiento que ofrece. Se brindan características específicas de la implementación para Phyton PySpark. Se presenta, además, una breve descripción de programas existentes en la UCLV para el cálculo de estas medidas, con el objetivo de especificar la transformación del cálculo de descriptores de proteínas en el Capítulo 2.

Capítulo 2. Implementación Spark del cálculo de descriptores de proteínas

En este capítulo se especifica el diseño empleado en la implementación paralela del software TI2BioP. Se muestra la propuesta de diseño del cálculo de cada descriptor utilizando Spark. Adicionalmente, se referencia el código fuente Spark de las funciones implementadas, así como los detalles de despliegue y ejecución en el clúster de Spark de la UCLV.

2.1 Diseño del programa paralelo TI2BioP

El programa TI2BioP calcula los descriptores K-mers, Spaced K-mers, Pseudo amino acid composition, Autocorrelation for Norm Moreau Broto, Autocorrelation for Moran, Autocorrelation for Geary Autocorrelation total, Composition descriptors (CTD_C), Transition descriptors (CTD_T), Distribution descriptors (CTD_D) y Quasi-sequence order descriptors a partir de un archivo FASTA que contiene el proteoma de una especie. Luego obtiene los resultados de cada descriptor en un archivo .csv separado por comas.

El archivo FASTA tiene la estructura siguiente: dos líneas para cada proteína, donde la primera línea comienza con el carácter < y seguidamente incluye el identificador de la proteína, y la segunda línea contiene la cadena de aminoácidos de un alfabeto de veinte de ellos. La

Figura 2. 1 muestra una sección de un archivo FASTA para un proteoma de levaduras (*Saccharomyces Castelli*) (Salichos and Rokas, 2011).

```
>Scas489.4
MSQEKKPVDPLHSFIAGALAGAIEASITYPFEFAKTRLQLIDKTSTASRNPLVLIYNTAKTQGTGAIYVGCPAFIVGNTAKAG
IRFLGFDTIKNMLRDPVTGELSGPRGVVAGLGAGLLESVVAVTPFEAIKTALIDDKQALKPKYQNNGRGMLRNYGSLVRD
QGIMGLYRGVLPVSMRQAANQAVRLGCYNKIKTMVQDYTNAPKDRPLSSGLTFIVGAFSGVVTVYTTMPIDTVKTRMQS
LDATKYTSTVNCFAKIFKEEGLKTFWKGATPRLGRLILSGGIVFTIYENVLVFLG
>Scas714.12
MTAELQQPKGNHRYQEPSQARMLKIPYNSTLSSGIRSFSVLSGLRATTIEPSNTNHPKSQSHLKNDEPTVKHANLSNAQ
KAYLDRAIRVDQAGELGANYIYAGQMFVLLRKHPHLKPILTHMWEQEVHHHNTFNALQLKNRVRPSLLTPFWKVGAIAM
GVTTAAISPEAAMACTEAVETVIGGHYNEQLRVLTNQFELDKTDGTRGVPKEVRKLINTIKEFRDQELEHLDTAIKNDSKE
AVPYKIITEGIKGICRIAVWSAERI
>Scas489.5
MSNLQHPESPSEDIELENSPQMNTHTSEINITQSDIPIDPRDTEQANEAEAETEEQTGSSTRDRLIPTELRERTTRQLGTL
GRRFNILDKIFGRKNPNQPQHPRGESYDGVFRNLAAKPESKRTRNAEGTDDTPPTYDEAAADMVPSYYGMDLSTSDMY
MDEICIEGLPVGNIANLLWNIIVSTSFQFIGFLITYILHTSHAAKQGSRFGLGLTFVGYAYSMIPNNVTTKVGKHKSISRIKLS
DPNSYDDVHLYSEPTTEDNFESTLSHGIDEEKQKVPFLAVAVGLLGLFISIKSIVDYIQVKKMEKRYLSQDQV
```

Figura 2. 1 Sección de archivo FASTA para un proteoma de levaduras.

La estructura general del programa para calcular los distintos descriptores se muestra en el diagrama de actividades de la Figura 2. 2. De acuerdo a la cantidad de procesadores de la computadora donde está desplegado el programa, se distribuye la cantidad de secuencias de aminoácidos por los distintos procesadores y cada uno efectúa los cálculos correspondientes para la sección de datos asignados a éste. Cada descriptor tiene varios argumentos que serán especificados en la siguiente sección donde se presenta la propuesta Spark de sus implementaciones.

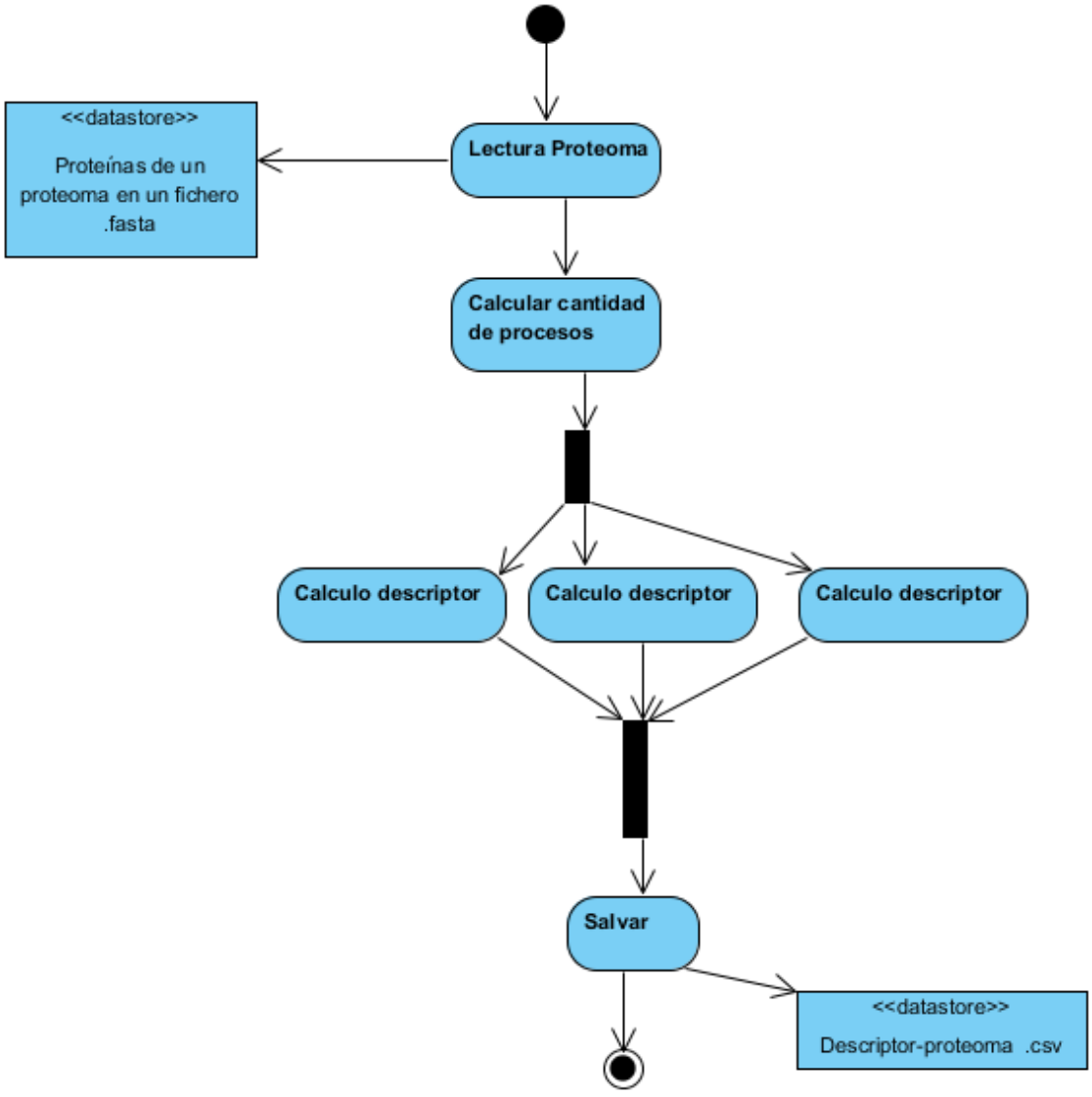

Figura 2. 2 Estructura general del cálculo paralelo de un descriptor.

En el caso específico del cálculo de la frecuencia de k-mers en una secuencia de proteínas, la función parte de un patrón base o semilla compuesto por las letras que componen las secuencias de proteínas (ACDEFGHIKLMNPQRSTVWY) y el tamaño de k deseado y genera todos los patrones de esas letras de ese tamaño mediante la

programación paralela. Luego carga el archivo .fasta y lo divide entre los procesadores con que cuente la máquina. En cada una de las divisiones ejecuta la función kmerscount que se encarga de contar la aparición de cada secuencia de tamaño k en una secuencia de proteína. Finalmente, une las frecuencias calculadas en paralelo para conformar el resultado completo. La función de k-mers espaciados funciona de manera similar excepto que en un paso intermedio paraleliza la formación de patrones de k-mers con espacios, donde no se incluyen espacios en los extremos. Algunos ejemplos de estos patrones serían: “101” para k-mers con k igual a 2 y un espacio, y “10101”, “10011” y “11001” para k-mers de tamaño 3 con dos espacios.

Una vez calculados los descriptores de las proteínas de cada proteoma, es posible realizar las comparaciones para pares de proteínas de dos proteomas siguiendo el esquema de paralelización que aparece en la Figura 2. 3. El programa correspondiente recibe como entrada dos ficheros de descriptores calculados para dos proteomas distintos, y de acuerdo al tamaño de la entrada distribuye el cálculo de una medida de similitud para los distintos pares en los distintos procesadores. En este diagrama de actividades se utiliza la correlación de Pearson (Deza, 2006) para la comparación par a par de los vectores del descriptor. El resultado de estas comparaciones es una matriz de similitud entre las proteínas de dos proteomas cuyos valores estarían normalizados. Esta matriz queda almacenada en un archivo .csv. En las implementaciones realizadas en Phyton con CUDA es necesario salvar información temporal según se realizaban los cálculos, cuestión que hace más lenta la ejecución.

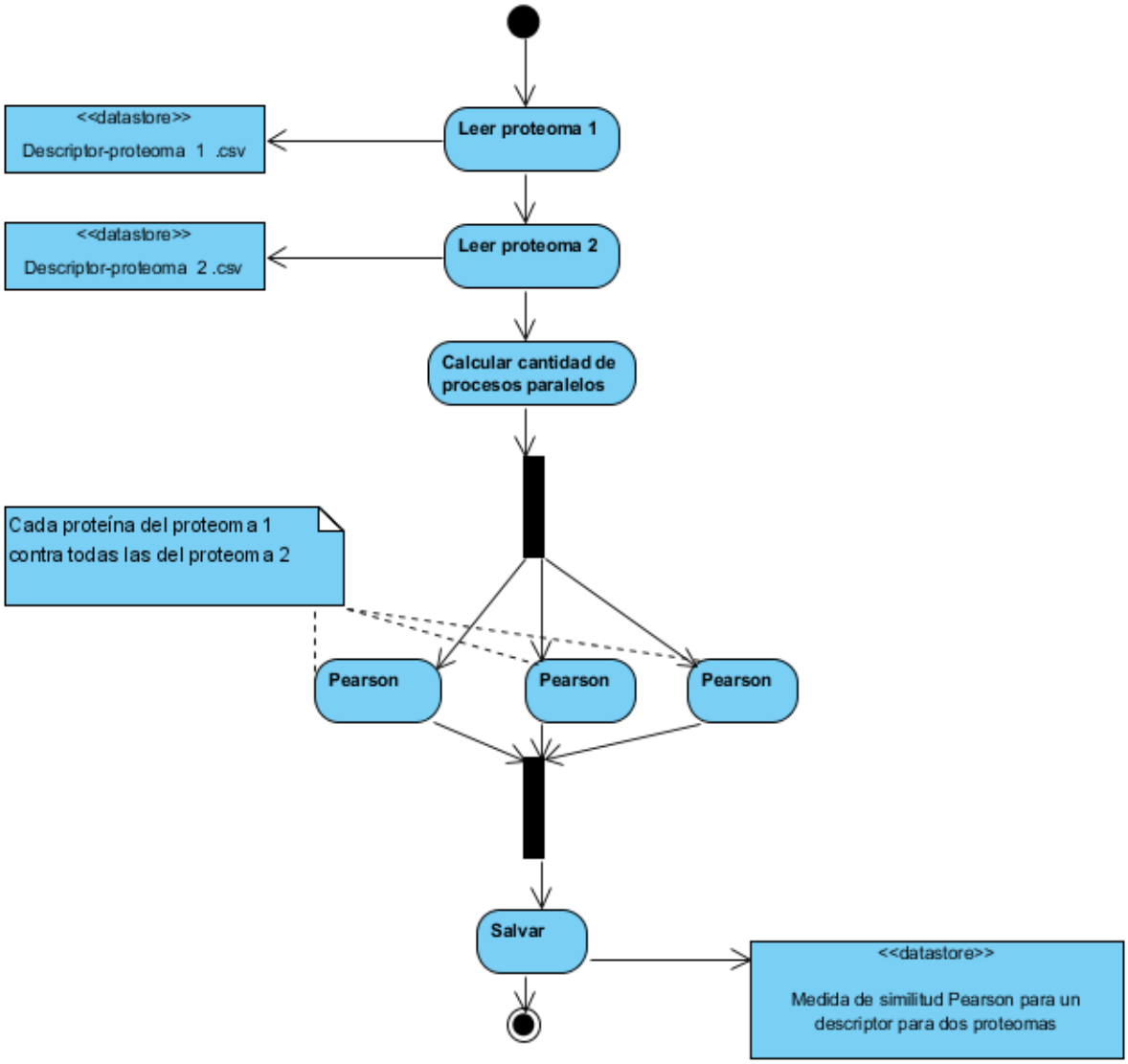

Figura 2. 3 Estructura general del cálculo paralelo de una medida de similitud para un descriptor.

2.2 Diseño del cálculo de descriptores utilizando PySpark

De manera general, todos los descriptores utilizan como entrada los archivos FASTA, ubicados en el sistema de archivos distribuidos HDFS. Luego estos descriptores distribuyen el cálculo en los distintos procesadores de los nodos del clúster mediante la transformación Map que ejecuta una función de cálculo de descriptor para una secuencia, previamente implementada en el programa TI2BioP. Finalmente, salvan el resultado del cálculo en archivos .csv dentro de una carpeta en el sistema HDFS. La Figura 2. 4 muestra el diagrama de actividades genérico para el cálculo de un descriptor para las proteínas de un proteoma.

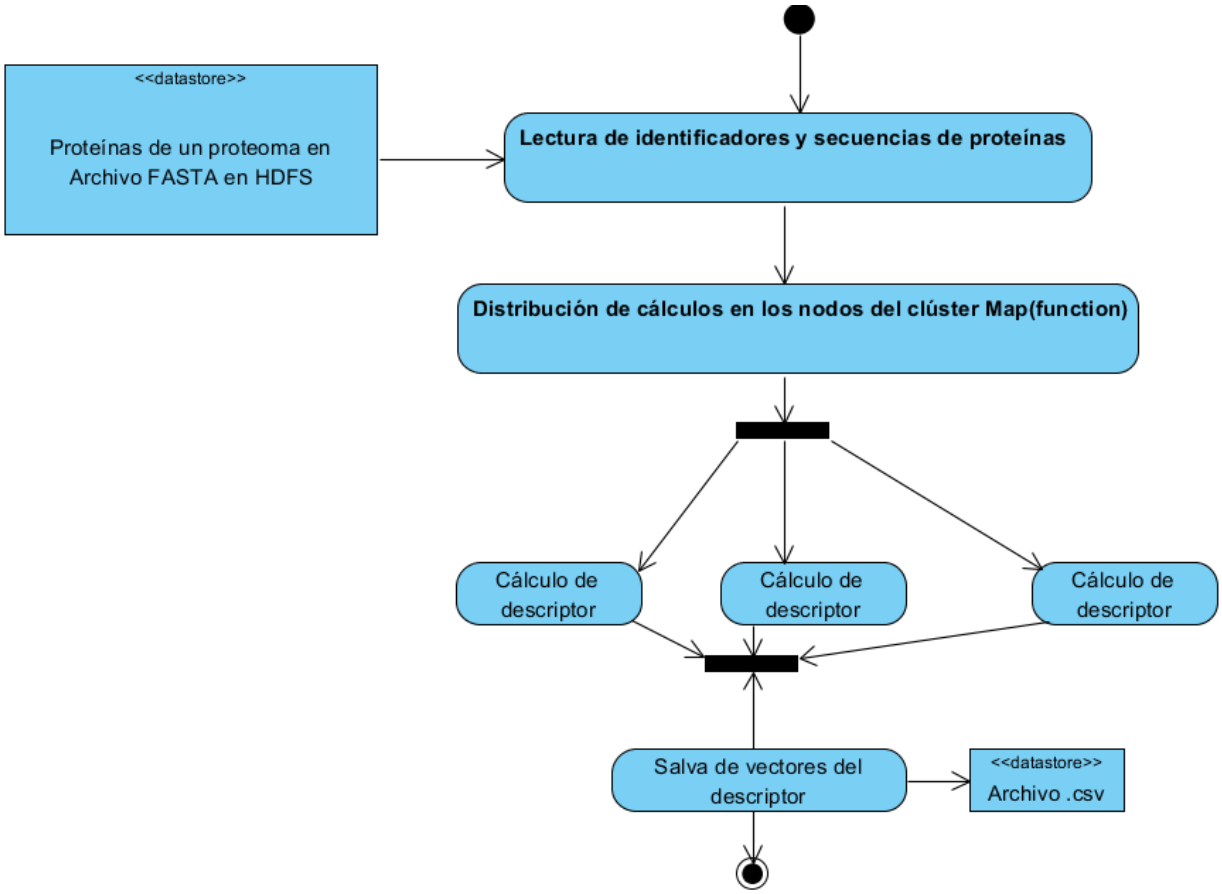

Figura 2. 4 Estructura general del cálculo Spark de un descriptor para las proteínas de un proteoma.

El flujo de creación de RDDs se especifica a continuación, igualmente de manera genérica indicando el orden de los pasos.

Inicialización del SparkContext. En esta inicialización es posible utilizar el valor local, ya que el valor enviado por el spark-submit es el del cluster master=yarn.

```
1     sc = SparkContext("local", "Descriptor")
```

Carga del archivo FASTA en un RDD1

```
2     logFile = args.fasta
3     nombreFASTA = logFile.split('.')[0]
4     f = open(logFile).read()
5     secuencias_id = f.split('>')
6     RDD1 = sc.parallelize([x.rstrip().replace('\n', ',') for
   x in secuencias_id if x != ''])
```

Creación de un RDD apareado con la estructura llave=idproteina, valor=secuencia

```
3    RDD2 = RDD1.map(lambda x:(x.split(',')[0],x.split(',')[1]))
```

Distribución del cálculo del descriptor para una secuencia

```
4    RDD3 = RDD2.map(lambda key:funcionCalculoDescriptor(key[0], key[1]))
```

Salva de los vectores como texto

```
5    RDD3.saveAsTextFile(nombreFASTA + '_Descriptor_results')
```

En caso de descriptores que requieren un número de aminoácidos como mínimo para el cálculo es posible utilizar un filtro entre el paso 2 y 3

filtro de secuencias con longitud mayor que 30

```
RDD3 = RDD2.filter(lambda keyValues: len(keyValues[1]) > 30)
```

El programa recibe un conjunto de argumentos y asume valores por defecto que se especifican a continuación:

```
description='Script for Protein Descriptors Calculations.'

'-k', type=int, help='K-mers size ( from 1 to 7)'

'-s', type=int, help='Space size ( from 1 to 4)'

parser.add_argument('-f', '--fasta', required=True, help='input file name'

'-l', '--lamda', type=int, help='Pseudo amino acid composition, (input Lamda)'

'-an', '--NMB', action='store_true', help='Autocorrelation for Norm Moreau Broto'

'-fs', '--filter', action='store_true', help='filter for small selection'

'-am', '--Moran', action='store_true', help='Autocorrelation for Moran'
```

```
'-ag', '--Geary', action='store_true', help='Autocorrelation for Geary'

'-at', '--Total', action='store_true', help='Autocorrelation total'

'-cc', '--CTD_C', action='store_true', help='Composition descriptors (CTD_C)'

'-ct', '--CTD_T', action='store_true', help='Transition descriptors (CTD_T)'

'-cd', '--CTD_D', action='store_true', help='Distribution descriptors (CTD_D)'

'-ctd', '--CTD', action='store_true', help='Composition, Transition, Distribution descriptors (CTD)'

'-qcn', '--QSOCN', action='store_true', help='Quasi sequence order coupling numbers '

'-qso', '--QSO', action='store_true', help='Quasi-sequence order descriptors'

'-m', '--maxlag', type=int, help='Maxlag (default 30)'

'-w', '--weight', type=float, help='Weight (default 0.1)'
```

En la Tabla 2. 1 se especifica el proceso de cálculo de cada descriptor

Tabla 2. 1 Descripción del cálculo de cada descriptor.

Descriptor	**Proceso de cálculo**
Pseudo composición de aminoácidos	Carga el archivo fasta y lo distribuye entre los procesadores de la máquina. Llama para cada uno de los fragmentos a la función calculePseudoAcc, la cual se encarga del cálculo llamando al método _GetPseudoAAC implementado en PseudoACC de la biblioteca Propy de Python. Luego guarda en un archivo texto.
Auto correlación	Norm Moreau Broto Autocorrelation Carga el archivo fasta y lo distribuye entre los procesadores de la máquina. Llama para cada uno de los fragmentos a la función calculeNormMoreauBroto que se encarga del cálculo de esta Auto correlación con el método CalculateNormalizedMoreauBrotoAuto

	implementado en Autocorrelation que se encuentra en la biblioteca Propy. Moran's Autocorrelation Llama a la función calculeMoran que calcula esta Auto correlación con el método CalculateMoranAuto implementado en Autocorrelation de la biblioteca Propy. Geary's Autocorrelation Llama a la función calculeGearyAuto que calcula esta Auto correlación con el método CalculateGearyAuto implementado en Autocorrelation de la biblioteca Propy. Total Autocorrelation Llama a la función calculateAutoTotal que se encarga del cálculo de esta Auto correlación con el método CalculateAutoTotal implementado en Autocorrelation de la biblioteca Propy de Python.
Composición, Transición y Distribución	Composition descriptor (CTD_C) Llama a la función calculateC que se encarga del cálculo de este descriptor mediante el método calculateC implementado en CTD de la biblioteca Propy. Transition descriptor (CTD_T) Llama a la función calculateT que calcula este descriptor mediante el método calculateT implementado en CTD de la biblioteca Propy de Python. Distribution descriptor (CTD_D) Llama a la función calculateD que calcula mediante el método calculateD implementado en CTD de la biblioteca Propy. Composition, Transition, Distribution descriptor (CTD) Llama a la función calculateCTD que se encarga del cálculo de este descriptor mediante el método calculate CTD implementado en CTD de la biblioteca Propy.
Quasi Sequence Order Descriptors	Quasi-sequence Order Coupling Number Llama a la función GetSequenceOrderCouplingNumberTotal que se encarga del cálculo de este descriptor mediante el método GetSequenceOrderCouplingNumberTotal implementado en QuasiSequenceOrder de la biblioteca Propy. Quasi-sequence order descriptor Llama a la función GetQuasiSequenceOrder que se encarga del

	cálculo de estos descriptores mediante el método GetQuasiSequenceOrder implementado en QuasiSequenceOrderde la biblioteca Propy.

k-mers	# Crea un RDD con todos los posibles k-mers a buscar en las secuencias y lo deposita en una variable broadcast accesible por todos los nodos del clúster.
	kmerspattern = sc.broadcast(makeMuster(k, alphabet))
	# Realiza la búsqueda de manera distribuida
	kmers = results.map(lambda key: kmercount(key[0], key[1], kmerspattern.value))

k-mers espaciados	Esta función ejecuta de forma similar a la de k-mers, excepto que crea patrones con espacios a partir de los posibles patrones a buscar formados por todos los aminoácidos. template = sc.broadcast(spacedk_mers(k, s)) newpatterns = purekmers.map(lambda x: setmuster(x, template.value)).collect() newmusters = sc.broadcast(newpatterns) kmers_spaced = results.map(lambda key: countKmers_spaced(newmusters.value, key[1], key[0]))

El código fuente completo de la implementación Spark se encuentra en el Anexo 1.

2.3 Despliegue y ejecución del cálculo de descriptores en el clúster de Spark de la UCLV

El clúster de big data de la UCLV es IBM iDataPlex d360 M2 y está compuesto por 30 nodos con dos procesadores Intel Xeon L5520 (8M Cache, 2.26 GHz, 5.86 GT/s Intel® QPI), 8 núcleos, 12 GB RAM, 2 x 1 Gbit Ethernet y una capacidad de almacenamiento de 148 GB (4.44 TB en general) cada uno. Su arquitectura se muestra en la Figura 2. 5 incluyendo un sistema de archivos distribuidos HDFS, un sistema de operación de datos YARN que permite la interacción entre Spark y HDFS, y el propio Spark entre otros componentes.

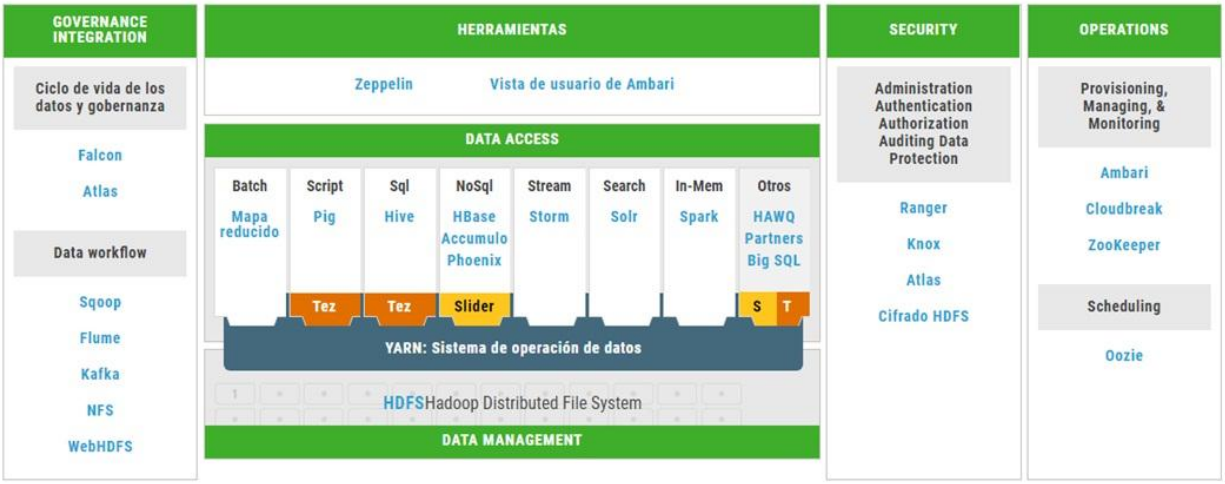

Figura 2. 5 Arquitectura del clúster de big data en la UCLV.

Para ejecutar el programa o aplicación en el clúster se debe utilizar el comando de Spark spark-submit (Karau et al., 2015) que de forma general se representa como:

spark-submit [options] <app jar | python file> [app options]

[options] Es una lista de banderas para spark-submit. Una lista de banderas comunes es enumerada la Tabla 2. 2 .

<app jar | python file> Se refiere al JAR o Script de Python conteniendo el punto de entrada hacia la aplicación.

[app options] Son opciones que serán pasadas a la aplicación. Si el método main () del programa llama a argumentos, se utiliza sólo [app options] y no las banderas específicas para spark-submit.

Tabla 2. 2 Banderas comunes para spark-submit.

Bandera	**Explicación**
--master	Especifica un URL de clúster para conectarse. Las opciones para esta bandera están descritas en la Tabla 2. 3

--deploy-mode	Para lanzar el programa del controlador localmente (“client”) o en una de las máquinas trabajadoras dentro del clúster (“cluster”). En spark-submit modo cliente el controlador ejecuta el programa en la misma máquina donde spark-submit está siendo invocado. En el modo clúster, el controlador envía para ejecutar en un nodo trabajador en el clúster. Por defecto está en modo cliente.
--class	La clase “principal” de la aplicación si se ejecutara un programa Java o Scala.
--name	Un nombre legible para la aplicación. Esto será exhibido en la web UI de Spark.
--jars	Una lista de archivos JAR para subir y colocar en el classpath de la aplicación. Si la aplicación depende en un pequeño número de JARs de terceros, se pueden añadir aquí.
--files	Una lista de archivos a ser colocados en el directorio de trabajo de su aplicación. Esto puede servir para archivos de datos a distribuir para cada nodo.
--py-files	Una lista de archivos a ser añadida al PYTHONPATH de la aplicación. Esta lista puede contener archivos .py, egg, o .zip.
--executor-memory	La cantidad de memoria a usar por los ejecutores, en bytes. Los sufijos pueden usarse para especificar mayores cantidades como “512m” (512 megabytes) o “15g” (15 gigabytes).
--driver-memory	La cantidad de memoria a usar para el proceso del controlador, en bytes. Los sufijos pueden usarse para especificar mayores cantidades como “512m” (512 megabytes) o “15g” (15 gigabytes).

Tabla 2. 3 Valores posibles para la bandera --master en spark-submit.

Valor	**Explicación**
spark://host:port	Conecta a un clúster de Spark Standalone en el puerto especificado. Por defecto los masters de Spark Standalone usan el puerto 7077.
mesos://host:port	Conecta a un master del clúster Mesos con el puerto especificado. Por defecto los masters de Mesos escuchan por el puerto 5050.
yarn	Conecta a un clúster de YARN. Al correr en YARN se necesita colocar la variable de ambiente HADOOP_CONF_DIR a apuntar la posición del directorio de configuración de Hadoop, el cual contiene la información acerca del clúster.
local	Ejecuta en modo local con un solo core.
local[N]	Ejecuta en modo local con N cores.
local[*]	Ejecuta en modo local y usa tantos cores como tenga la máquina.

En el caso específico del script BigD_Descriptor.py el spark-submit se ejecuta de la siguiente forma

```
spark-submit \

--master  yarn \

--deploy-mode cluster \

--name "ProtDescriptors" \
```

2.4 Consideraciones finales del Capítulo

En este capítulo se analiza la versión paralela actual del cálculo de descriptores con sus inconvenientes en cuanto a la salva de archivos temporales y la limitación de ejecución en una máquina con CUDA. Se presenta una propuesta de implementación en Spark por las ventajas especificadas en el Capítulo 1. Se demuestra cómo es posible lograr la implementación reutilizando el código Python de la biblioteca Propy para el cálculo de los distintos descriptores y a su vez aprovechando las potencialidades de Spark como el uso de RDD apareados por llave, valor. La implementación PySpark resulta factible para el acceso a archivos en un sistema de

archivos distribuidos como HDFS y para lograr la ejecución en la configuración establecida en el cluster de la UCLV. En el siguiente capítulo se mostrará una experimentación preliminar con el programa BigD_Descriptor.py.

Capítulo 3. Experimentación preliminar del cálculo de descriptores de proteínas

En este capítulo se muestran los resultados experimentales preliminares realizados con un proteoma de levadura. Por la disponibilidad de nodos en el clúster de Spark de la UCLV en el momento de tomar los datos de la experimentación se utilizó un clúster de Spark standalone instalado en el Centro de Bioactivos Químicos de la propia Universidad. Se trabajó en modo local con un único nodo con doce procesadores. No obstante, el programa fue desarrollado en el clúster de la UCLV donde se realizaron las primeras pruebas con resultados alentadores.

La Figura 3. 1 muestra una captura de pantalla de la ejecución en el clúster standalone utilizado con una línea de comandos empleada en combinación con el comando time para recoger los tiempos de ejecución.

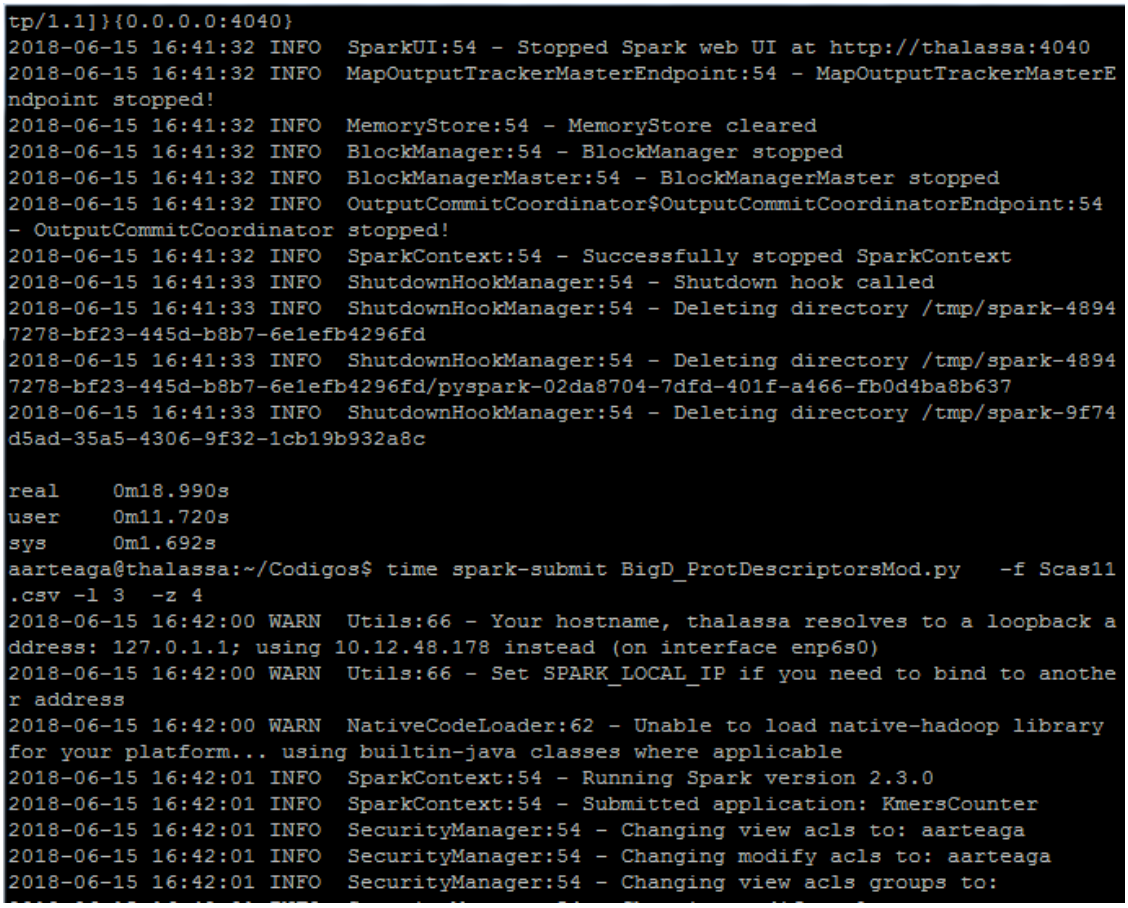

Figura 3. 1 Captura de pantalla de la ejecución del programa BigD_ProtDescriptor.py.

3.1 Conjuntos de datos

Para la experimentación preliminar se utilizó el proteoma de una levadura *Saccharomyce Castelli* con 4681 proteínas (Salichos and Rokas, 2011). El resto de los proteomas de levaduras seleccionados para los experimentos aparecen en la dirección **/user/aarteaga/FicherosEntrada** del servicio HDFS del cluster de Spark de la UCLV como se muestra en la Figura 3. 2. En esta figura un archivo aparece con el texto ”mod” en el nombre ya que ha sido modificado con la estructura de un .csv, es decir, separado por comas cada identificador de proteína y su respectiva secuencia.

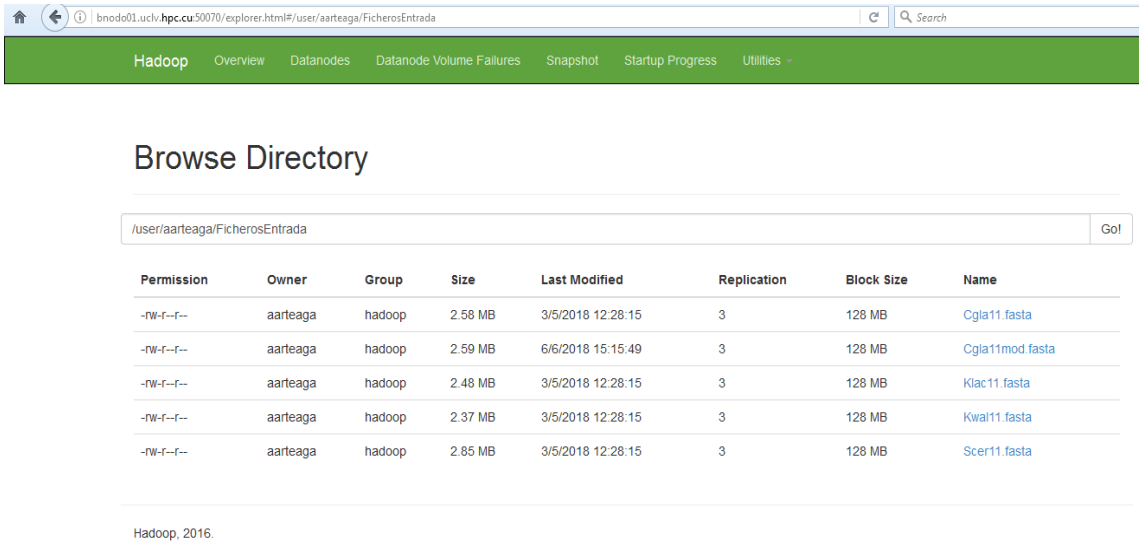

Figura 3. 2 Archivos fasta en la carpeta HDFS.

Para depositar los datos en dicha carpeta se utilizó un comando como el que sigue:

```
hadoop fs -put -f Cgla11mod.fasta /user/aarteaga/FicherosEntrada
```

Los archivos de salida aparecen en la carpeta **/user/aarteaga/FicherosSalida** en carpetas separadas para cada descriptor y su estructura aparece en la Figura 3. 3.

	A	B	C	D	E	F	G	H	I	J	K	L
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	0.000000	0.000000	0.000000	1,000000	0.000000	1,000000	1,000000	2,000000	0.000000	0.000000	0.000000	0.000000
3	0.000000	0.000000	0.000000	0.000000	0.000000	1,000000	0.000000	0.000000	1,000000	0.000000	0.000000	0.000000
4	2,000000	0.000000	0.000000	0.000000	0.000000	1,000000	0.000000	1,000000	0.000000	0.000000	0.000000	1,000000
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1,000000	1,000000
6	0.000000	0.000000	0.000000	0.000000	1,000000	0.000000	0.000000	0.000000	0.000000	2,000000	0.000000	1,000000
7	0.000000	0.000000	0.000000	0.000000	0.000000	1,000000	0.000000	0.000000	0.000000	1,000000	0.000000	1,000000
8	0.000000	0.000000	1,000000	1,000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
9	0.000000	0.000000	0.000000	1,000000	0.000000	0.000000	1,000000	0.000000	0.000000	0.000000	0.000000	0.000000

Figura 3. 3 Estructura de los archivos de salida en la carpeta HDFS.

3.2 Descriptores y valores de parámetros

La Tabla 3. 1 contiene los valores de los parámetros utilizados en la experimentación para cada descriptor.

Tabla 3. 1 Descriptores empleados en la experimentación con sus valores de parámetros.

Descriptor	**Parámetros y valores**
Pseudo composición de aminoácidos	$\lambda = 3$
Pseudo composición de aminoácidos	$\lambda = 4$
K-mers	$k = 2$
K-mers	$k = 3$
K-mers	$k = 4$
K-mers/espaciado	$k = 2\ \ s = 1$
K-mers/espaciado	$k = 2\ \ s = 2$
K-mers/espaciado	$k = 2\ \ s = 3$
K-mers/espaciado	$k = 3\ \ s = 1$
K-mers/espaciado	$k = 3\ \ s = 2$

Descriptor	**Parámetros y valores**
K-mers/espaciado	$k = 3 \ s = 3$
Auto correlación de Geary	-
Auto correlación de Moran	-
Auto correlación Total	-
Composición, Distribución y Transición (Composición) (CTD_C)	-
Composición, Distribución y Transición (Distribución) (CTD_D)	-
Composición, Distribución y Transición (Transición) (CTD_T)	-
Composición, Distribución and Transición (Total) (CTD)	-
Quasi-Sequence-Order (QSO)	maxlag = 30 weight=0.1
Quasi-Sequence-Order Coupling Numbers (QSOCN)	maxlag = 30

En el descriptor de composición de k-mers, k representa el tamaño de palabras contiguas (posiciones donde hay correspondencia), y en el descriptor k-mers espaciado se especifica k como la cantidad de posiciones coincidentes y s la cantidad de posiciones ocupadas por cualquier aminoácido. Para $k = 2$, con posición de correspondencia (1) y una posición de cualquier carácter (0), el patrón a utilizar sería: “101”. Para $k = 2$ con dos posiciones de cualquier carácter, el patrón sería: “1001”. Para $k = 3$ con tres posiciones de cualquier carácter, el patrón sería: “10001”. De igual forma, con $k = 3$ con una posición de cualquier carácter, los patrones serían: “1101”, “1011”. Con $k = 3$ con dos posiciones de cualquier carácter, los patrones serían:

10011", "10101", "11001". Finalmente, con $k = 3$ con tres posiciones de cualquier carácter, los patrones serían: "100011", "110001", "101001", "100101".

3.3 Tiempos de ejecución

En la Tabla 3. 2 se muestran los datos del proteoma elegido para hacer las mediciones. Se ha aplicado un filtro para reducir la muestra de secuencias de acuerdo a su longitud. Las secuencias deben tener más de 30 aminoácidos para calcular algunos de los descriptores por lo que establece esta longitud de secuencias como mínima para la ejecución del programa.

Tabla 3. 2 Archivo fasta utilizado en la medición con cantidad de secuencias antes y después de un filtrado realizado.

Archivo fasta	Cantidad de Secuencias	Filtro	Cantidad de Secuencias después del filtrado
Scas11	4681	Tamaño de secuencia >30 y <100	120
		Tamaño de secuencia >30	4678

Las siguientes tablas (Tabla 3. 3, Tabla 3. 4, Tabla 3. 5, Tabla 3. 6, Tabla 3. 7, Tabla 3. 8) muestran los tiempos de ejecución de cada descriptor en minutos y segundos para diferentes valores de parámetros. En estas tablas las dos primeras filas representan las dos primeras ejecuciones donde se utilizó la configuración local con un procesador. Luego para las dos filas siguientes se utilizaron doce procesadores. Aparecen en negrita señalados los mejores tiempos para cada configuración de procesadores y en negrita y subrayado el mejor tiempo en la tabla.

En la Tabla 3. 3 de tiempo de ejecución del descriptor k-mers se puede observar de manera general que se reduce el tiempo con el aumento en la cantidad de procesadores, aunque no siempre esto se cumple. Por otra parte, el aumento tanto en el valor de k como en la cantidad de secuencias y su longitud tienden a influir en el aumento del tiempo de ejecución.

La Tabla 3. 4 de tiempo de ejecución del descriptor k-mers espaciado también muestra en las dos primeras filas las ejecuciones en un procesador y en las dos últimas, las ejecuciones con doce procesadores. De igual forma se han señalado en negrita los

mejores tiempos para cada configuración y subrayado el mejor tiempo para el descriptor de k-mer espaciado. Los mejores tiempos se observan para k=2 con dos espacios y para k=3 con un espacio, siendo este último tiempo el menor logrado al incorporar doce procesadores. La longitud del k-mer o del k-mer espaciado, conjuntamente con la cantidad de espacios permitidos parece influir en el aumento del tiempo de ejecución.

Tabla 3. 3 Tiempo de ejecución del descriptor k-mers con tres valores de k que son: 2, 3 y 4.

Archivo fasta/ procesadores	Filtro	K-mer		
		K=2	K=3	K=4
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	**0m5.966s**	0m6.653s	0m21.755s
	Tamaño de secuencia >30	0m8.830s	0m52.897s	16m9.229s
Scas11/ 12 procesadores	Tamaño de secuencia >30 y <100	0m7.825s	0m9.323s	0m14.267s
	Tamaño de secuencia >30	**0m7.373s**	0m29.819s	7m58.790s

Tabla 3. 4 Tiempo de ejecución del descriptor k-mers espaciado.

Archivo fasta	Filtro	K-mer					
		K=2 S=1	K=2 S=2	K=2 S=3	K=3 S=1	K=3 S=2	K=3 S=3
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	0m26.989s	**0m12.845s**	0m13.423s	1m51.390s	2m30.184s	
	Tamaño de secuencia >30	2m5.883s	2m6.088s	2m8.543s	56m42.311s	79m17.772s	
Scas11/ 12 procesadores	Tamaño de secuencia >30 y <100	0m20.998s	0m18.312s	0m10.898s	**0m7.985s**	1m9.355s	1m35.908s
	Tamaño de secuencia >30	1m1.957s	1m4.572s	1m5 .696s	28m38.875s	39m24.321s	54m15.309s

En la Tabla 3. 5 aparece el tiempo de ejecución para descriptores de Auto correlación. El menor tiempo se obtiene para la Auto correlación de Geary para uno y doce procesadores siendo esta última configuración la que arroja en general los menores tiempos para los tres tipos de descriptores de Auto correlación.

Por otra parte, en la Tabla 3. 6 se muestra el tiempo de ejecución para los descriptores de Composición, Distribución y Transición siendo el descriptor CTD el más rápido con un procesador y el CTD_D con doce procesadores. De igual forma el descriptor QSOCN obtuvo los menores tiempos con las dos configuraciones de procesadores como se observa en la Tabla 3. 7.

Considerando los valores de λ=3 y λ=4, el descriptor de pseudo composición de aminoácidos muestra los mejores valores para λ=3 con uno y doce procesadores, como se observa en la Tabla 3. 8 con los valores de las ejecuciones que se pudieron realizar. Al aumentar los cálculos para conformar un vector descriptor de más longitud según aumenta el valor de λ, este parámetro puede influir en el tiempo de ejecución.

Analizando los resultados de manera general se observa que el tiempo de ejecución en la mayoría de las corridas se reduce con el aumento en el total de procesadores disponibles. El menor tiempo global (0m4.755s) se obtuvo con el descriptor QSOCN para 4678 secuencias con doce procesadores, mientras que el mayor tiempo (108m52.510s) se registra en el cálculo de la pseudo composición de aminoácidos para λ=3 considerando 4678 secuencias. Este tiempo se reduce aproximadamente a la mitad (52m45.816s) al utilizar doce procesadores. De forma similar se comporta el cálculo de k-mers espaciados (k=3, s=2) donde el mayor tiempo de 79m17.772s se reduce a (39m24.321s) con doce procesadores y el siguiente mayor tiempo (56m42.311s) con (k=3, s=1) se reduce a (28m38.875s). Se debe señalar que el 70% de los tiempos registrados están por debajo de un minuto mientas que el resto fue reducido con los doce procesadores. La Tabla 3. 9 recoge la distribución de mediciones por debajo y por encima de un minuto considerando nombres genéricos de descriptores. En negrita aparece resaltada la cantidad de mediciones mayores de un minuto para el descriptor k-mer espaciado.

Tabla 3. 5 Tiempo de ejecución de los descriptores de Auto correlación (de Moran, de Geary y Total).

Archivo fasta	Filtro	Auto correlación de Moran	Auto correlación de Geary	Auto correlación Total
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	0m6.132s	**0m6.077s**	0m8.434s
	Tamaño de secuencia >30	0m24.940s	0m26.674s	7m14.012s
Scas11/ 12 procesadores	Tamaño de secuencia >30 y <100	0m6.179s	**0m6.008s**	0m7.176s
	Tamaño de secuencia >30	0m15.739s	0m15.857s	3m35.927s

Tabla 3. 6 Tiempo de ejecución de descriptores de Composición, Distribución y Transición.

Archivo fasta	Filtro	CTD_C	CTD_D	CTD_T	CTD
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	0m6.475s	0m6.082s	0m6.095s	**0m6.046s**
	Tamaño de secuencia >30	0m7.278s	0m16.008s	0m7.594s	0m19.071s
Scas11/ 12 procesadores	Tamaño de secuencia >30 y <100	0m5.891s	**0m5.944s**	0m5.959s	0m5.954s
	Tamaño de secuencia >30	0m6.560s	0m11.009s	0m6.713s	0m12.298s

Tabla 3. 7 Tiempo de ejecución de descriptores QSO.

Archivo fasta	Filtro	QSO	QSOCN
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	0m7.126s	0m5.230s
	Tamaño de secuencia >30	1m43.690s	**0m4.853s**
Scas11/ 12 procesadores	tamaño de secuencia >30 y <100	0m6.169s	0m4.869s
	tamaño de secuencia >30	0m54.317s	**0m4.755s**

Tabla 3. 8 Tiempo de ejecución del descriptor de pseudo composición de aminoácidos.

Archivo fasta	Filtro	Pseudo composición de aminoácidos	
		λ= 3	λ= 4
Scas11/ 1 procesador	Tamaño de secuencia >30 y <100	**0m31.086s**	0m31.441s
	Tamaño de secuencia >30	108m52.510s	72m8.910s
Scas11/ 12 procesadores	Tamaño de secuencia >30 y <100	**0m18.990s**	
	Tamaño de secuencia >30	52m45.816s	

Tabla 3. 9 Distribución de tiempos menores y mayores que un minuto considerando nombres genéricos de descriptores.

Descriptor	Tiempo de ejecución menor de 1 min	Tiempo de ejecución mayor de 1 minuto
k-mers	10	2
k-mers espaciados	7	**15**
Auto correlación	10	2
Descriptores de composición, Composición, Distribución y Transición	16	0
Quasi-Sequence-Order	7	1
Pseudo composición de aminoácidos	3	3
Total de mediciones	53	23
Por ciento del total	70%	30%

3.4 Consideraciones finales del capítulo

La experimentación preliminar realizada arroja resultados alentadores en cuanto a la reducción del tiempo de ejecución en los descriptores que más demoran como el de pseudo composición de aminoácidos y el de k-mers espaciados al aumentar la cantidad de procesadores. Adicionalmente, se detecta una variación en los tiempos según varían los valores de los parámetros en estos descriptores. A su vez, los experimentos confirman la necesidad de ampliar las pruebas para más proteomas y diferentes configuraciones de nodos y procesadores con vistas a estudiar cómo se comportará el programa ante mayores requerimientos de entrada de datos y mayor cantidad de recursos a utilizar.

Conclusiones

1. Entre las variantes de modelos de programación analizadas para su aplicación en el cálculo de descriptores de proteínas considerando la configuración del clúster de computadoras de la Universidad Central “Marta Abreu” de Las Villas, el Spark resulta ser una selección adecuada por ser parte de dicha configuración para ejecutar programas distribuidos y presentar facilidades para el manejo a alta velocidad de grandes volúmenes de datos con uso intensivo de memoria, tolerancia a fallas, entre otras.
2. El diseño de funciones del cálculo de descriptores de proteínas mediante PySpark se basa en el modelo Spark y a su vez reutiliza códigos disponibles en bibliotecas de Phyton como Propy y BioPhyton.
3. Las implementaciones de las funciones del cálculo de descriptores utilizando el modelo Spark se ejecutan en un clúster de Spark con acceso al sistema de archivos distribuidos hdfs, por lo que puede ser aplicado al cálculo de descriptores de múltiples proteínas en múltiples proteomas.
4. Las pruebas realizadas a las funciones implementadas arrojan resultados alentadores en cuanto a la reducción del tiempo de ejecución al aumentar la cantidad de procesadores en los descriptores que consumen más tiempo como el de pseudo composición de aminoácidos y el de k-mers espaciados.
5. Los diversos valores de los parámetros, así como la cantidad de secuencias y el tamaño de las mismas, conjuntamente con los recursos computacionales que se emplean pueden influir en el tiempo de ejecución de las implementaciones realizadas en Spark para los distintos descriptores.

Recomendaciones

1. Ampliar las pruebas del cálculo de descriptores al clúster de la UCLV con todas sus prestaciones.
2. Implementar en Spark el cálculo de similitud por pares utilizando las funciones de cálculo de descriptores implementadas.
3. Implementar en Spark otros descriptores de proteínas.
4. Aplicar los cálculos a múltiples proteomas y medir los resultados de escalabilidad de los cálculos.

Referencias bibliográficas

AGÜERO-CHAPIN, G., MOLINA-RUIZ, R., MALDONADO, E., DE LA RIVA, G., SÁNCHEZ-RODRÍGUEZ, A., VASCONCELOS, V. & ANTUNES, A. 2013 Jul 16. Exploring the adenylation domain repertoire of nonribosomal peptide synthetases using an ensemble of sequence-search methods. *PloS one,* 8**,** e65926.

AGÜERO-CHAPIN, G., PÉREZ-MACHADO, G., MOLINA-RUIZ, R., PÉREZ-CASTILLO, Y., MORALES-HELGUERA, A., VASCONCELOS, V. & ANTUNES, A. 2011 Feb 1. TI2BioP: Topological Indices to BioPolymers. Its practical use to unravel cryptic bacteriocin-like domains. *Amino acids,* 40**,** 431-42.

BHASIN, M. & RAGHAVA, G. P. S. 2004. Classification of Nuclear Receptors Based on Amino Acid Composition and Dipeptide Composition. *J. Bio. Chem.,* 279**,** p.23262.

BONVIN, N. 2012. *Linear Scalability of Distributed Applications.* ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE.

CAO, D.-S., XU, Q.-S. & LIANG, Y.-Z. 2013. propy: a tool to generate various modes of Chou's PseAAC. *Bioinformatics,* 29**,** 960-962.

CATTANEO, G., PETRILLO, U. F., GIANCARLO, R. & ROSCIGNO, G. 2015. Alignment-Free Sequence Comparison over Hadoop for Computational Biology. *In:* IEEE (ed.) *2015 44th International Conference on Parallel Processing Workshops.* Conference Publishing Services.

CHOU, K.-C. 2000. Prediction of protein subcellular locations by incorporating quasi-sequence-order effect. *Biochemical and biophysical research communications,* 278**,** 477-483.

CHOU, K.-C. 2001. Prediction of Protein Cellular Attributes Using Pseudo-Amino Acid Composition. *PROTEINS: Structure, Function, and Genetics,* 43**,** 246–255.

DEZA, E. 2006. *Dictionary of Distances*, Elsevier.

DUBCHAK, I., MUCHNIK, I., HOLBROOK, S. R. & KIM, S. H. 1995. Prediction of protein folding class using global description of amino acid sequence. *Proc Natl Acad Sci U S A,* 92, 8700-8704.

DUBCHAK, I., MUCHNIK, I., MAYOR, C., DRALYUK, I. & KIM, S. H. 1999. Recognition of a protein fold in the context of the SCOP classification. *Proteins: Structure, Function, and Bioinformatics,* 35**,** 401-407.

ELLOUMI, M. & ZOMAYA, A. Y. (eds.) 2011. *Algorithms In Computational Molecular Biology* Hoboken, New Jersey: John Wiley & Sons, Inc.

ESPINOSA, J. Q. 2015. *Massive Scale Multimodal Data.* Master en ingeniería informática, Universi Politècnica De Cataluny.

FERNÁNDEZ, A., RÍO, S. D., LÓPEZ, V., BAWAKID, A., JESUS, M. J. D., BENÍTEZ, J. M. & HERRERA, F. 2014. Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks. *WIREs Data Mining and Knowledge Discovery,* 4**,** 380-409.

GALPERT, D. 2016. *Contribuciones al enfoque de comparación par a par en la detección de genes ortólogos.* Doctor en Ciencias Técnicas, Universidad Central "Marta Abreu" de las Villas.

GALPERT, D., FERNÁNDEZ, A., HERRERA, F., ANTUNES, A., MOLINA-RUIZ, R. & AGÜERO-CHAPIN, G. 2018. Surveying alignment-free features for ortholog detection in related yeast proteomes by using supervised big data classifiers. *BMC Bioinformatics*.

GALPERT, D., GARCÍA, S. D. R., HERRERA, F., ANCEDE-GALLARDO, E., ANTUNES, A. & AGÜERO-CHAPIN, G. 2017. Big Data Supervised Pairwise Ortholog Detection in Yeasts. *Yeast - Industrial Applications.* IntechOpen.

GALPERT, D., RÍO, S. D., HERRERA, F., ANCEDE-GALLARDO, E., ANTUNES, A. & AGÜERO-CHAPIN, G. 2015. An Effective Big Data Supervised Imbalanced Classification Approach for

Ortholog Detection in Related Yeast Species. *BioMed Research International* [Online], 2015, Article ID 748681.

GOYA, A., GALPERT, D., MILLO, R. & COMPANIONI, C. 2016. Análisis de la escalabilidad del cálculo paralelo de medidas de similitud entre pares de genes. *Revista Cubana de Ciencias Informáticas,* 10**,** 71-84.

GUNASINGHE, U., ALAHAKOON, D. & BEDINGFIELD, S. 2014. Extraction of high quality k-words for alignment-free sequence comparison. *Journal of Theoretical Biology,* 358**,** 31-51.

HORNE, D. S. 1988 Mar. Prediction of protein helix content from an autocorrelation analysis of sequence hydrophobicities. *Biopolymers,* 27**,** 451-77.

KAMVYSSELIS, M. K. 2003. *Computational comparative genomics: genes, regulation, evolution.* Doctor of Philosophy in Computer Science, Massachusetts Institute of Technology

KARAU, H., KONWINSKI, A., WENDELL, P. & ZAHARIA, M. 2015. Learning Spark. *In:* BEAUGUREAU, A. S. A. M. (ed.) *Lightning-Fast Data Analysis.* United States of America: O'Reilly Media, Inc.

KARAU, H. & WARREN, R. 2017. High Performance Spark: Best Practices for Scaling & Optimizing. O'Reilly Media, Inc.

KASHYAP, H., AHMED, H. A., HOQUE, N., ROY, S. & BHATTACHARYYA, D. K. 2014. Big Data Analytics in Bioinformatics: A Machine Learning Perspective. *JOURNAL OF LATEX CLASS FILES,,* VOL. 13.

KRISTENSEN, D. M., WOLF, Y. I., MUSHEGIAN, A. R. & KOONIN, E. V. 2011. Computational methods for Gene Orthology inference. *Briefings in bioinformatics,* 12**,** 379-391.

KUMAR, M., THAKUR, V. & RAGHAVA, G. P. 2008. COPid: composition based protein identification. *In silico biology,* 8**,** 121-128.

LANDABURO-DEL-ARCO, G., GALPERT, D. & JORGE, A. G. 2016. *Aplicación del modelo de programación MapReduce al cálculo de medidas de similitud para pares de genes.* Ingeniería Informática Trabajo de Diploma, Universidad Central "Marta Abreu" de Las Villlas.

LEIMEISTER, C.-A., BODEN, M., HORWEGE, S., LINDNER, S. & MORGENSTERN, B. 2014. Fast alignment-free sequence comparison using spaced-word frequencies. *Bioinformatics,* 30**,** 1991-1999.

LIN, J. 2013. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything Thats Not a Nail! *Big Data,* 1**,** 28-37.

MAHMOOD, K., WEBB, G. I., SONG, J., WHISSTOCK, J. C. & KONAGURTHU, A. S. 2012. Efficient large-scale protein sequence comparison and gene matching to identify orthologs and co-orthologs. *Nucleic Acids Research,* 40.

MATSUNAGA, A., TSUGAWA, M. & FORTES, J. 2008. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. *Fourth IEEE International Conference on eScience.* IEEE Computer Society.

MILLO, R. & GALPERT, D. 2012. *Aplicación de medidas de similitud y algoritmos de agrupamiento a la detección de genes ortólogos.* Trabajo de Diploma, Universidad Central "Marta Abreu" de Las Villas.

MOLINA, R., AGÜERO-CHAPIN, G. & PÉREZ-GONZÁLEZ, M. 2011. TI2BioP (Topological Indices to BioPolymers) version 2.0. *Molecular Simulation and Drug Design (MSDD), Chemical Bioactives Center, Central University of Las Villas, Cuba*.

NICKOLLS, J., BUCK, I. & GARLAND, M. 2008. Scalable Parallel Programming with CUDA. ACM QUEUE.

PENTREATH, N. 2015. Real-time Machine Learning with Spark Streaming.

RAO, H. B., ZHU, F., YANG, G. B., LI, Z. R. & CHEN, Y. Z. 2011. Update of PROFEAT: a web server for computing structural and physicochemical features of proteins and peptides from amino acid sequence. *Nucleic Acids Research,* 39**,** W385-W390.

SALICHOS, L. & ROKAS, A. 2011. Evaluating ortholog prediction algorithms in a yeast model clade. *PLoS ONE,* 6**,** e18755.

SNIR, M. & OTTO, S. (eds.) 1998. *MPI-The Complete Reference: The MPI Core,* Cambridge, MA: MIT Press.

SOKAL, R. R. & THOMSON, B. A. 2006 Jan. Population structure inferred by local spatial autocorrelation: an example from an Amerindian tribal population. *Am J Phys Anthropol,* 129**,** 121-31.

SONNHAMMER, E. L. L., GABALDÓN, T., SILVA, A. W. S. D., MARTIN, M., ROBINSON-RECHAVI, M., BOECKMANN, B., THOMAS, P. D. & DESSIMOZ, C. 2014. Big data and other challenges in the quest for orthologs. *Bioinformatics Editorial***,** 1-6.

VINGA, S., AND JONAS ALMEIDA 2003. Alignment-free sequence comparison—a review. *Bioinformatics,* 19**,** 513-523.

WALL, D. P., KUDTARKAR, P., FUSARO, V. A., PIVOVAROV, R., PATIL, P. & TONELLATO, P. J. 2010. Cloud computing for comparative genomics. *BMC Bioinformatics,* 11**,** 259-259.

ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S. & STOICA, I. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *9th USENIX Conference on Networked Systems Design and Implementation.* San Jose, CA.

ZUNDA-HERRERA, L. A. & GALPERT, D. 2016. *Utilización del Weka-Spark para la clasificación de genes ortólogos.* Licenciatura en Ciencia de la Computación Trabajo de Diploma, Universidad Central "Marta Abreu" de Las Villlas.

Anexo 1. Código fuente del programa BigD_Descriptor.py

```
from Bio import SeqIO
from itertools import product
import numpy as np
from Propy import Autocorrelation as AC
from Propy import PseudoAAC as PAAC
from Propy import CTD as CTD
from Propy import QuasiSequenceOrder as QSO
from pyspark import SparkConf, SparkContext
import argparse
import csv
import os
import sys
import re


# Autocorrelations
# Autocorrelation modules

def calculeNormMoreauBroto(id, proteinSequence):
    temp                                                                =
AC.CalculateNormalizedMoreauBrotoAuto(proteinSequence,
[AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
    return result


def calculeMoran(id, proteinSequence):
    temp      =      AC.CalculateMoranAuto(str(proteinSequence),
[AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
    return result


def calculeGearyAuto(id, proteinSequence):
    temp      =      AC.CalculateGearyAuto(str(proteinSequence),
[AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
    return result


def calculateAutoTotal(id, proteinSequence):
    temp = AC.CalculateAutoTotal(str(proteinSequence))
    # result = temp['ResidueASA']
    temp['name'] = id
    return temp
```

```
# SOCN: sequence order coupling numbers (depend on the choice of
maxlag, the default is 60)
# QSO: quasi-sequence order descriptors (depend on the choice of
maxlag, the default is 100)

def   getSequenceOrderCouplingNumberTotal(id,   proteinSequence,
maxlag=30):
    temp                                                          =
QSO.GetSequenceOrderCouplingNumberTotal(str(proteinSequence),
maxlag)
    temp['name'] = id
    return temp


def   getQuasiSequenceOrder(id,   proteinSequence,   maxlag=30,
weight=0.1):
    temp    =    QSO.GetQuasiSequenceOrder(str(proteinSequence),
maxlag, weight)
    temp['name'] = id
    return temp


# modules for CTD
# CTD: Composition, Transition, Distribution descriptors (CTD)
(21+21+105=147)

def calculateC(id, proteinSequence):
    temp = CTD.CalculateC(str(proteinSequence))
    temp['name'] = id
    return temp


def calculateT(id, proteinSequence):
    temp = CTD.CalculateT(str(proteinSequence))
    temp['name'] = id
    return temp


def calculateD(id, proteinSequence):
    temp = CTD.CalculateD(str(proteinSequence))
    temp['name'] = id
    return temp


def calculateCTD(id, proteinSequence):
    temp = CTD.CalculateCTD(str(proteinSequence))
    temp['name'] = id
    return temp


# Kmers calculations
def spacedk_mers(k, l):
    result = []
    istring = k * 'X' + l * '.'
    c = [''.join(x) for x in product(istring, repeat=k + l)]
    for i in c:
```

```
        if i.count('X') == k:
            if (i.find('.') != 0) and (i.rfind('.') != k + l -
1):
                result.append(i)
    result = list(set(result))
    return result


def setmuster(patronelement, musters):
    modmusters = []
    modmusters.append(patronelement)
    for muster in musters:
        cad = ''
        counter = 0
        for j in range(len(muster)):
            if muster[j] == 'X':
                cad = cad + patronelement[counter]
                counter += 1
            else:
                cad = cad + '.'
        modmusters.append(cad)
    return modmusters


def countKmers_spaced(listakms, wheretolook, id):
    kmersview = {}
    kmersview['name'] = id
    counter = 0
    for xlists in listakms:
        for kms in xlists:
            kmersview[kms]           =           re.findall(kms,
wheretolook).__len__()
    return kmersview


def makeMuster(k, alphabet):
    """
     k: mustter size
     alphabet : mustter context
    :return:
    """
    return [''.join(x) for x in product(alphabet, repeat=k)]


def kmercount(id, seq, patrones):
    """ seq : Sequence
        patrones: mutter to look for
        id: id for sequence
        :return a dictionary patronDict
    """
    patronDict = {}
    patronDict['name'] = id
    for k in patrones:
        patronDict[k] = seq.count(k)
    return patronDict
```

```
# Pseudo Amino acid composition

def calculePseudoAcc(id, proteinSequence, lamdasize):
    # return a dictionary with PseudoAAC
    if len(proteinSequence) > lamdasize:
        temp      =      PAAC._GetPseudoAAC(str(proteinSequence),
lamdasize, weight=0.05)
        temp['name'] = id
    else:
        temp = {}
        c = ['pac{}'.format(i) for i in range(21 + lamdasize)]
        d = c[1:]
        temp['name'] = id
        for x in d:
            temp[x] = 0.0

    return temp


# Arguments for ProteinDescriptors.py
parser = argparse.ArgumentParser(description='Script for Protein
Descriptors calculations.')
parser.add_argument('-k', type=int, help='K-mers size ( from 1
to 7)')
parser.add_argument('-s', type=int, help='Space size ( from 1 to
4)')
parser.add_argument('-f', '--fasta', required=True, help='input
file name')
parser.add_argument('-l',   '--lamda',   type=int,   help='Pseudo
amino acid composition, (input Lamda)')
parser.add_argument('-an',     '--NMB',     action='store_true',
help='Autocorrelation for Norm Moreau Broto')
parser.add_argument('-fs',    '--filter',    action='store_true',
help='filter for small selection')
parser.add_argument('-am',    '--Moran',    action='store_true',
help='Autocorrelation for Moran')
parser.add_argument('-ag',    '--Geary',    action='store_true',
help='Autocorrelation for Geary')
parser.add_argument('-at',    '--Total',    action='store_true',
help='Autocorrelation total')
parser.add_argument('-cc',    '--CTD_C',    action='store_true',
help='Composition descriptors (CTD_C)')
parser.add_argument('-ct',    '--CTD_T',    action='store_true',
help='Transition descriptors (CTD_T)')
parser.add_argument('-cd',    '--CTD_D',    action='store_true',
help='Distribution descriptors (CTD_D)')
parser.add_argument('-ctd', '--CTD', action='store_true',
                    help='Composition,  Transition,  Distribution
descriptors (CTD)')
parser.add_argument('-qcn',    '--QSOCN',    action='store_true',
help='Quasi sequence order coupling numbers ')
parser.add_argument('-qso',     '--QSO',     action='store_true',
help='Quasi-sequence order descriptors')
parser.add_argument('-m',   '--maxlag',   type=int,   help='Maxlag
(default 30)')
```

```
parser.add_argument('-w', '--weight', type=float, help='Weight
(default 0.1)')
parser.add_argument('-z', '--running', type=str, help='running
number')

if __name__ == "__main__":
    k = 0  # Kmers
    s = 0  # Spaced
    l = 0  # lamda for Pseudo amino acid composition
    alphabet = 'ACDEFGHIKLMNPQRSTVWY'  # Protein alphabet
    ejecutar = False
    calcule = False
    calcule2 = False
    calcule3 = False

    args = parser.parse_args()
    if args.k:
        k = int(args.k)
        calcule = True
    if args.s:
        s = int(args.s)
        calcule = True

    if args.lamda:
        l = int(args.lamda)
        calcule2 = True

    if args.maxlag:
        maxlag = int(args.maxlag)
    else:
        maxlag = 30

    if args.weight:
        weight = float(args.weight)
    else:
        weight = 0.1

    # input checking

    if k > 7 or k < 0:
        parser.print_help()
        print ('Error input for kmers ... k_{}'.format(k))
        sys.exit()

    if s < 0 or s > 4:
        parser.print_help()
        print       ('Error       input       for       spaced
kmers......s_{}'.format(s))
        sys.exit()

    if l < 0:
        parser.print_help()
        print('Error input for Lamda ......l_{}'.format(l))
        sys.exit()

    logFile = args.fasta
```

```
    splitname = logFile.split('.')[0]
    # Initializing SparkContext
    sc = SparkContext("local", "KmersCounter")
    # load text file as RDD
    f = open(logFile).read()
    secuencias_id = f.split('>')
    logData = sc.parallelize([x.rstrip().replace('\n', ',') for
x in secuencias_id if x != ''])
    # build a pair RDD
    fasta    =    logData.map(lambda    x:    (x.split(',')[0],
x.split(',')[1]))
    print('{} secuencias cargadas'.format(fasta.count()))
    # filter applied to get small sequences

    if args.filter:
        results      =      fasta.filter(lambda      keyValues:
len(keyValues[1]) > 30 and len(keyValues[1]) < 100)
        checkseq        =        results.map(lambda        key:
len(key[1])).collect()
        np.savetxt(splitname    +    '_seqdata.txt',    checkseq,
delimiter=',')
    else:
        results      =      fasta.filter(lambda      keyValues:
len(keyValues[1]) > 30)

    print('{}          Sequences          to          calculate
...'.format(results.count()))


    if calcule:
        if k != 0 and s != 0:
            # Calculation of spaced_kmers
            # build mustter to search into sequences
            purekmers = sc.parallelize(makeMuster(k, alphabet))
            kmerspattern = sc.broadcast(makeMuster(k, alphabet))
            template = sc.broadcast(spacedk_mers(k, s))
            newpatterns = purekmers.map(lambda x: setmuster(x,
template.value)).collect()
            newmusters = sc.broadcast(newpatterns)
            kmers_spaced     =     results.map(lambda     key:
countKmers_spaced(newmusters.value, key[1], key[0]))
            print('Spaced                kmers                {}
founds'.format(kmers_spaced.count))
            kmers_spaced.saveAsTextFile(splitname   +   '_k_'   +
str(k)  +  '_s_'  +  str(s)  +  '_Kmers_spaced_results_run_'  +
args.running)
        elif k != 0 and s == 0:
            # calculation of kmers
            kmerspattern = sc.broadcast(makeMuster(k, alphabet))
            # perfom searching
            kmers = results.map(lambda key: kmercount(key[0],
key[1], kmerspattern.value))
            kmers.saveAsTextFile(splitname + '_k_' + str(k) +
'_Kmers_results_run_' + args.running)

    if calcule2:
```

```
        if l > 0:
            psedoAAC       =       results.map(lambda       key:
calculePseudoAcc(key[0], key[1], l))
            psedoAAC.saveAsTextFile(splitname                    +
'_PseudoAC_lambda_' + str(l) + '_results_run_' + args.running)

    if args.NMB:
        normMoreauBroto      =      results.map(lambda      key:
calculeNormMoreauBroto(key[0], key[1]))
        normMoreauBroto.saveAsTextFile(splitname                 +
'_Norm_MoreauBroto_results_run_' + args.running)

    if args.Moran:
        moranAuto = results.map(lambda key: calculeMoran(key[0],
key[1]))
        moranAuto.saveAsTextFile(splitname                       +
'_MoranAuto_results_run_' + args.running)

    if args.Geary:
        gearyAuto        =        results.map(lambda        key:
calculeGearyAuto(key[0], key[1]))
        gearyAuto.saveAsTextFile(splitname                       +
'GearyAuto_results_run_' + args.running)

    if args.Total:
        xtotalAuto        =        results.map(lambda        key:
calculateAutoTotal(key[0], key[1]))
        xtotalAuto.saveAsTextFile(splitname                      +
'_TotalAuto_results_run_' + args.running)

    # Composition, transition, distribution descriptors (CTD)

    if args.CTD_C:
        xctdc   =   results.map(lambda   key:   calculateC(key[0],
key[1]))
        xctdc.saveAsTextFile(splitname + '_CTD_C_results_run_' +
args.running)

    if args.CTD_T:
        xctdt   =   results.map(lambda   key:   calculateT(key[0],
key[1]))
        xctdt.saveAsTextFile(splitname + '_CTD_T_results_run_' +
args.running)

    if args.CTD_D:
        xctdd   =   results.map(lambda   key:   calculateD(key[0],
key[1]))
        xctdd.saveAsTextFile(splitname + '_CTD_D_results_run_' +
args.running)

    if args.CTD:
        xctd   =   results.map(lambda   key:   calculateCTD(key[0],
key[1]))
        xctd.saveAsTextFile(splitname  +  '_CTD_results_run_'  +
args.running)
```

```
    if args.QSO:
        qso          =          results.map(lambda          key:
getQuasiSequenceOrder(key[0],       key[1],       maxlag=maxlag,
weight=weight))
        qso.saveAsTextFile(splitname    +    '_QSO_maxlag_'    +
str(maxlag)  +  'weight_'  +  str(weight)  +  '_results_run_'  +
args.running)

    if args.QSOCN:
        # QSO Calculations and save to directory
        qsoresults       =       results.map(lambda       key:
getSequenceOrderCouplingNumberTotal(key[1], key[0]))
        qsoresults.saveAsTextFile(splitname                    +
'_QSOCN_results_run_' + args.running)

    sc.stop()
```