HMusket: corrector de secuencias mediante el espectro k-mer basado en Hadoop

Luis Lorenzo Mosquera
GAC (Grupo de Arquitectura de Computadores)
Dpto. de Ingeniería de Computadores
A Coruña, Spain
luis.lorenzom@udc.es

***Resumen*—La alta demanda de análisis de datos genéticos a lo largo de la última década ha incrementado la necesidad de herramientas paralelas que puedan procesar todo este volumen da datos en tiempos razonables. Una de las fases más presentes en este tipo de análisis es la corrección de secuencias, ya que durante la amplificación de las muestras a secuenciar se pueden introducir errores debido a fallos del ADN Polimerasa. A lo largo de este proyecto se presenta el diseño e implementación de HMusket, una herramienta Big Data para sistemas de memoria distribuida implementada con Hadoop mediante el modelo MapReduce. HMusket logra acelerar la corrección de grandes volúmenes de datos hasta 30 veces frente a una aplicación análoga de memoria compartida. Además de la herramienta se presenta un análisis de las distintas soluciones disponibles actualmente. Esta herramienta está disponible en https://github.com/luislorenzom/hmusket**

***Index Terms*—Big Data, MapReduce, Hadoop, Corrector de secuencias**

I. Introducción

Debido a la aparición de las tecnologías conocidas como *Next Generation Sequence* (NGS) [1] es posible obtener en poco tiempo grandes volúmenes de datos genéticos procedentes de diversos seres vivos (humanos, animales, plantas, etc). Estos vastos conjuntos de datos se utilizan en diferentes aplicaciones bioinformáticas como el alineamiento de secuencias a genomas de referencia o análisis de variantes genómicas. No obstante, el continuo abaratamiento de estas tecnologías permite a los científicos realizar análisis más exhaustivos donde tienen más cabida todos estos datos. Uno de los objetivos más ambiciosos que ya se están alcanzando con estos datos en el área de las ciencias de la vida (biología, medicina, etc.) es la predicción de enfermedades de carácter o predisposición genética en un estadio temprano para evitar futuras complicaciones en el paciente.

Otro campo donde se utilizan de forma activa los conjuntos de datos NGS es la metagenómica (el estudio desde el punto de vista genómico de comunidades microbianas). Este amplio campo ha adquirido gran importancia en los últimos años ya que permite descubrir el fraude o las malas condiciones en la industria alimentaria, determinar las condiciones medioambientales de una zona e incluso detectar ciertas enfermedades por los patógenos registrados en una muestra. Por último, la farmacogenética, disciplina que consiste en la interacción de la información genética adquirida durante la fase de secuenciación junto con una base de conocimiento de las áreas de la farmacología y patología, puede predecir qué fármaco es más efectivo para un paciente, incluso dar lugar al diseño de uno nuevo.

No obstante, para llegar a tales fines y obtener unos resultados confiables son necesarios una serie de pasos previos. Además de obtener una muestra de ADN y secuenciarla, es necesario corregir los datos obtenidos tras la secuenciación, ya que durante la fase de amplificación o síntesis de nuevas copias por medio de técnicas PCR [2] es posible que se incorporen errores debido a algún fallo del ADN Polimerasa [3]. Afortunadamente esta clase de fallos siguen un proceso estocástico y pueden ser corregidos en su mayoría a través de diversos algoritmos.

La gran demanda de datos genéticos que se está produciendo a lo largo de la última década, tiene como consecuencia que las soluciones tradicionales que corrigen estos errores produzcan un cuello de botella en los estudios anteriormente mencionados. Esto es debido a que la mayoría de las herramientas paralelas para la corrección de errores están desarrolladas para sistemas de memoria compartida. Por lo tanto, su escalabilidad esta limitada por una sola máquina y no logran reducir considerablemente los tiempos de preprocesado de los datos.

El propósito de este trabajo es de proveer a los científicos con una herramienta paralela denominada HMusket para sistemas de memoria distribuida que pueda reducir el tiempo de corrección de grandes conjuntos de datos genéticos. Para la implementación de dicha herramienta resulta de interés la aplicabilidad de las tecnologías Big Data como MapReduce, las cuales pueden proporcionar los mecanismos adecuados para el procesamiento de grandes volúmenes de datos. Además de presentar la herramienta, es de interés detallar las tecnologías utilizadas a lo largo del proyecto (Sección II), analizar que otras alternativas hay en el mercado para llevar a cabo tal fin ya sean de memoria compartida o distribuida (Sección III), comentar el diseño de la herramienta y los entresijos de la implementación (sección IV) y finalizar con un análisis experimental (Sección V), junto con una serie de conclusiones y trabajo futuro (Sección VI).

II. Conocimiento previo

A continuación se describen brevemente las diversas tecnologías utilizadas en este proyecto.

II-A. MapReduce

Modelo de programación paralelo propuesto por Google [4] que surge ante la necesidad de procesar cantidades ingentes de datos de forma distribuida. Este paradigma consta de dos operaciones fundamentales derivadas de la programación funcional: Map y Reduce.

La operación Map convierte un par (clave, valor) en otro conjunto intermedio de datos en el mismo formato de tupla. Dicho formato hace mucho más eficiente el procesado de los datos y una futura reconstrucción de los mismos. Puede verse un ejemplo de esta operación en el Listing 1.

Listing 1: Ejemplo de operación Map

```
# Se inicializa un dataset con unos valores
dataset = [1, 2, 3, 4, 5]
# Se realiza la operacion de elevar al
# cuadrado, mediante una funcion lambda
# y el resultado se almacena como una lista
dataset = list(map(lambda x: x**2, dataset))
# dataset = [1, 4, 9, 16, 25]
```

Por otro lado, la operación Reduce utiliza los conjuntos de datos, ya sean intermedios generados por las operaciones Map o los datos originales, para agruparlos y obtener un resultado final (véase Listing 2 para un ejemplo de código Reduce).

Listing 2: Ejemplo de operación Reduce

```
# Se inicializa un dataset con unos valores
dataset = [1, 2, 3, 4, 5]
# Al igual que en el ejemplo anterior
# se utiliza una funcion lambda, en este caso
# la multiplicacion de dos numeros
value = reduce((lambda x, y: x * y), dataset)
# value = 120
```

II-B. Apache Hadoop

Hadoop [5] es un framework de código abierto desarrollado en Java orientado a la ejecución de aplicaciones distribuidas en un entorno clúster y al procesamiento paralelo y eficiente de grandes conjuntos de datos. Hasta la fecha Hadoop es la implementación más utilizada y popular del modelo MapReduce.

II-C. Hadoop Distributed File System (HDFS)

HDFS [6] es un sistema de ficheros distribuido que permite a las aplicaciones del ecosistema Hadoop trabajar con una alta tolerancia a fallos. Además, facilita el acceso a los datos que se encuentran repartidos entre todo el conjunto de computadores que componen el clúster Hadoop. Es relevante destacar que este sistema de ficheros no cumple en su totalidad con el estándar POSIX, lo cual provoca incompatibilidades con la mayoría del software existente.

II-D. Hadoop Sequence Parser (HSP)

Uno de los principales inconvenientes que tiene trabajar con Hadoop es la entrada de parámetros tanto en las clases Map como en las Reduce. Por defecto, Hadoop ofrece formatos de entrada para tipos de dato comúnmente usados, como por ejemplo texto, valores numéricos, etc. Esto quiere decir que los formatos más utilizados para almacenar las secuencias (fastq/fasta) no pueden ser procesados de una manera directa. Como solución a este problema, no solo para este proyecto sino para otros, se desarrolló una librería para el procesamiento de conjuntos de datos de secuencias fasta/fastq que se almacenan en HDFS. De esta manera se evita un preprocesado de los datos al inicio del programa con el consecuente cuello de botella que ello implica.

II-E. Java Native Interface (JNI)

JNI [7] es una librería que permite que un determinado código Java ejecutado sobre la máquina virtual envíe y reciba llamadas desde código nativo, es decir, programas y/o librerías desarrolladas en C, C++ o ensamblador.

III. Trabajos relacionados

A lo largo de esta sección se presentarán las diferentes soluciones que actualmente hay disponibles para realizar la tarea de corrección de secuencias, además de un breve comentario acerca de su funcionalidad y/o limitaciones.

III-A. Memoria compartida

Dentro del grupo de soluciones que explotan los sistemas de memoria compartida cabe destacar las soluciones basadas en GPU como son CUDE-EC [8] y DecGPU [9]. Estas herramientas utilizan un algoritmo basado en la realización de lecturas cortas del genoma, lo que no provee una solución completa de errores ni una alta precisión en los resultados (no se da cobertura total a todas las regiones genómicas). No obstante, al estar desarrolladas para GPUs presentan una escalabilidad muy alta debido a la alta capacidad computacional de estos dispositivos.

Otra solución que utiliza un algoritmo basado en lecturas cortas del genoma, pero usando CPU en lugar de GPU, es SOAP Corrector [10]. Recientes versiones de este programa utilizan, en unas determinadas operaciones de su pipeline, un método basado en el grafo De Brujin. Esto permite reducir drásticamente el uso de memoria en casos donde el tamaño del genoma pueda presentar problemas.

Siguiendo con las soluciones que están basadas en modelos de grafos se encuentra Reptile [11]. Este software hace uso de un grafo Hamming, el cual se combina junto con el análisis del espectro k-mer, para resolver las posibles ambigüedades que se encuentren en el genoma o región genómica a corregir, muy útil en casos que presenten errores de translocación. Sin ser un software basado en un modelo de grafos, SGA [12] también consigue optimizar el uso de la memoria utilizando la trasformada de Burrows-Wheeler y el FM-Index para representar el espectro k-mer de la región genómica.

Además de los métodos de lecturas cortas y de utilización de grafos hay alternativas basadas en modelos probabilísticos como, por ejemplo, Quake [13]. Este corrector utiliza la probabilidad acumulada de los k-mer que conforman el genoma para poder clasificar si se trata de un error o no.

Otros programas combinan soluciones ya mencionadas como pueden ser algoritmos basados en grafos y modelos probabilísticos. Un ejemplo es Hammer [14], que se compone de una solución basada en un grafo Hamming y un modelo probabilístico similar al que implementa Quake.

Musket [15], es un software que permite la corrección de datos genómicos en base al espectro k-mer del mismo. Este programa desarrollado en C++ está paralelizado mediante la API OpenMP [16] pudiendo acelerar parte de su pipeline de corrección de datos entre los distintos cores de la máquina donde se esta ejecutando. Para realizar dicha aceleración, los desarrolladores decidieron abordar esta problemática utilizando un patrón maestro/esclavo, el cual permite minimizar problemas relacionados con el desbalanceo de carga e hilos ociosos. Además de lo ya citado anteriormente, este corrector ofrece gracias al uso del análisis del espectro k-mer una gran cobertura de todo el código genético procesado.

Por último, cabe destacar que algunos correctores hacen uso de arrays de sufijos, como por ejemplo HiTEC [17], el cual instancia el genoma con distintos k-mers para posteriormente construir esos arrays y analizar los posibles errores. SHREC [18] es otra herramienta que utiliza un método parecido a HiTEC pero para la detección de indels [19] y sustituciones.

Según la literatura reciente [20], Musket ha demostrado ser la herramienta que mejores resultados proporciona en relación a la corrección de código y la cobertura del mismo, obteniendo un Valor-F del 81 % corrigiendo un conjunto de datos de una cobertura media, mientras que el resto de herramientas fallaban en medio del proceso o no alcanzaban la misma precisión.

III-B. Memoria distribuida

Dada la reciente necesidad de un post-procesado masivo de datos apenas hay soluciones de memoria distribuida para la corrección de errores. Solo destacan una solución en este paradigma.

CloudRS [21], es una herramienta implementada para el entorno Hadoop mediante el paradigma MapReduce. Esta herramienta requiere de un preprocesado del conjunto de secuencias de entrada antes de copiarlo a HDFS, lo cual causa un cuello de botella de considerables dimensiones por el intenso tráfico de E/S a disco. HMusket consigue eliminar dicho preprocesado gracias al uso de la librería HSP [22] como se ha mencionado en la sección II-D.

IV. Diseño e implementación

En líneas generales la motivación de este proyecto es distribuir entre varios nodos un determinado conjunto de datos genéticos, ya sea fasta o fastq en modo single-end o paired-end, y ejecutar varias instancias de Musket de forma simultánea. Finalmente se debe realizar una operación de unión o *merge* de todas las salidas obtenidas en el clúster. El principal motivo para utilizar Musket como corrector subyacente en lugar de otros similares (e.g. Reptile) es porque, como ya se citó anteriormente, la literatura reciente demuestra que este es el corrector con mayor precisión hasta la fecha.

Para el desarrollo del presente trabajo se tuvieron que completar dos hitos que componen la totalidad del proyecto. En posteriores párrafos se detallan en profundidad, pero en esencia dichos objetivos fueron la creación de una librería compartida para poder invocar la ejecución del algoritmo de Musket desde Java, y la creación de una aplicación distribuida mediante Hadoop denominada HMusket. De esta manera se pueden dividir los conjuntos de datos de entrada entre los distintos nodos de cómputo para posteriormente ejecutar la librería que contiene el algoritmo de corrección de secuencias.

No obstante, antes de detallar el diseño e implementación de HMusket conviene dar una visión general de como están estructuradas las aplicaciones Hadoop.

IV-A. Estructura general de una aplicación Hadoop

Las aplicaciones Hadoop, por lo general, consisten en un programa driver donde se configura la aplicación (ficheros de entrada, salida, formatos con los que va a trabajar, etc.). A su vez, en el driver se instancia un objeto de la clase Job a ejecutar en el clúster, al cual se le indican con qué clases Map y Reduce debe trabajar y el orden de las mismas. La implementación de dichas clases Map y Reduce siguen el siguiente ciclo de vida o flujo:

- Setup: operación que se realiza al inicio de la etapa Map o Reduce
- Map o Reduce: operación/función que se realiza por cada par (clave, valor) del conjunto de datos de entrada.
- Cleanup: operación que se realiza al final de la etapa Map o Reduce.

IV-B. Desarrollo de HMusket

HMusket sigue la estructura indicada en el apartado anterior. En el driver se recogen los parámetros de entrada que se reciben al lanzar la aplicación (e.g. ruta al fichero de entrada fasta/fastq). Posteriormente, se invoca al método parse de la clase CLIParse (véase Figura 1) el cual evalúa la obligatoriedad de dichos parámetros, configura parte del driver y genera una cadena que conforma los argumentos que recibirá la librería de Musket para ejecutar el algoritmo de corrección de secuencias.

Una vez configurado el driver, se establecen los ficheros de entrada y salida de los mappers: tanto el que procesa los conjuntos de datos single-end como el de los paired-end, dependiendo del caso. Para ello se hace uso de las implementaciones de las clases InputFormat de Hadoop que proporciona la librería HSP. Esto es necesario para que Hadoop sepa qué tipo de información está procesando cada tarea Map. Es decir, establecer si las secuencias de entrada que esta recibiendo son de tipo fasta, donde se procesa una cadena para la secuencia con las bases y otra para el identificador, o fastq, que a mayores de la cadena de las bases y el identificador, proporciona la opción de añadir un comentario y una segunda cadena que informa de la calidad de las bases. De esta manera cada mapper recibe una parte proporcional del conjunto de datos de entrada, evitando así que, por ejemplo, un mapper reciba de entrada secuencias o cadenas de calidad desparejadas

o tener que realizar un preprocesado del conjunto de datos de entrada al inicio de la aplicación y como consecuencia de esto generar cuellos de botella. El único requerimiento para que este método funcione es subir previamente el conjunto de datos genéticos a HDFS, algo inherente al uso de cualquier aplicación Hadoop.

Las distintas clases Map que se incluyen en HMusket instancian durante la fase setup un objeto de tipo PrintWriter: uno en el caso de conjuntos single-end y dos para los conjuntos paired-end. Posteriormente, en la etapa Map, se escribe toda la información de entrada al disco duro local del nodo de cómputo. Esto es necesario porque la librería que ejecuta Musket utiliza E/S mediante sistemas de ficheros basados en POSIX, mientras que para poder distribuir y poder trabajar adecuadamente con Hadoop se tiene que utilizar el sistema de ficheros HDFS.
Durante la etapa cleanup se cierra el buffer de escritura de los objetos PrintWriter y se efectúa la llamada al código nativo de Musket, pasándole por parámetro la cadena que conforma los argumentos necesarios generado previamente en la clase CLIParse. Tras la ejecución de Musket, se eliminan los archivos locales que se utilizaron para su entrada y se copian sus salidas a HDFS. Finalmente, se realiza la operación merge de todas las salidas de cada mapper para obtener un único fichero con la salida final.

La nula referencia a clases Reduce, tal y como se puede apreciar en la Figura 1 y en los párrafos anteriores, es debido a que para el diseño de HMusket no es necesario la ejecución de una etapa Reduce, ya que solo interesa dividir el conjunto de datos y aplicar la función corregir de Musket sobre cada partición.

Listing 3: Cabecera JNI generada

```
/*
 * Class:  es_udc_gac_hmusket_MusketCaller
 * Method: call
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
    Java_es_udc_gac_hmusket_MusketCaller_call
    (JNIEnv *, jobject, jstring);
```

IV-C. Creación e integración de la librería

Para poder realizar la llamada al código nativo de Musket desde Java es necesario convertir la herramienta en una librería compartida para su uso desde JNI. Posteriormente esa librería debe ser instalada en el clúster Hadoop para que cualquier aplicación del ecosistema pueda hacer uso de la misma.
Para efectuar esta tarea hay que crear una clase dentro del proyecto HMusket, en este caso fue MusketCaller, donde se especifique que durante la fase de instanciación se cargue la librería nativa. Además de esta indicación se tiene que crear un método con la etiqueta "native", que al igual que en los métodos abstractos tan solo se debe especificar la firma.
A continuación utilizando el comando "javac" proporcionado por el entorno de desarrollo de Java, se compila dicha clase generando un fichero .class. Este fichero es requerido para generar el fichero de cabeceras (.h) por medio del comando "javah" para posteriormente darle una implementación. El contenido del fichero de cabeceras se puede ver en el Listing 3. Dicha implementación debe ser incluida en un fichero C o C++ donde se implemente la firma indicada en el fichero de cabeceras, y por último realizar la llamada a la aplicación Musket.
Para poder realizar esta llamada como si fuera una función de una librería en vez de un ejecutable, se requiere una única modificación en el Makefile de Musket para añadir los flags del compilador necesarios para crear una librería compartida ("-fPIC" y "-shared"). Una vez compilada, la librería se añade al directorio $HADOOP_HOME/lib/native y por último se declarara la función main, del proyecto Musket, dentro del código C desarrollado en HMusket.

Tabla I: Especificaciones de un nodo de cómputo

Modelo CPU	2 × Intel Xeon E5-2660 Sandy Bridge-EP
Velocidad/Turbo CPU	2.20 GHz/3 GHz
Cores por CPU	8
Threads por core	2
Cores físicos/virtuales	16/32
Cache L1/L2/L3	32 KB/256 KB/20 MB
Memoria RAM	64 GB DDR3 1600 Mhz
Discos	1 × HDD 1 TB SATA3 7.2K rpm
Redes	InfiniBand FDR y Gigabit Ethernet

V. Evaluación experimental

En primer lugar se detallará el entorno, tanto software como hardware, donde se realizaron las pruebas para maximizar la reproducibilidad de los experimentos que se muestran a lo largo de esta sección. Posteriormente se analizará el rendimiento de Musket para conocer la horquilla de tiempos susceptibles de ser mejorados por HMusket. Para concluir, se expondrán los experimentos, resultados y conclusiones acerca de HMusket.

V-A. Entorno de pruebas

Para la realización de las diversas pruebas expuestas a lo largo de los siguientes párrafos se utilizó la cabina 0 del clúster Plutón [23] de la Facultad de Informática de la Universidade da Coruña. Las características de los nodos de cómputo se muestran en la Tabla I.

El sistema operativo que se ejecuta en estos servidores es Rocks 6.1, una distribución para entornos clúster basada en CentOS 6.9, mientras que el kernel utilizado es 2.6.32-696.23.1.el6.x86_64. La maquina virtual de Java utlizada ha sido OracleJDK 1.7.0_80. Por último, Musket ha sido compilado usando la suite de GNU en su versión 6.3.1.

Para la realización de pruebas en el clúster Pluton se utilizo la versión 3.1 de BDEv [24]. Por medio de esta herramienta se puede desplegar un clúster Hadoop con todo su ecosistema de aplicaciones, pudiéndolo configurar de manera sencilla simplemente con tener una versión del JRE

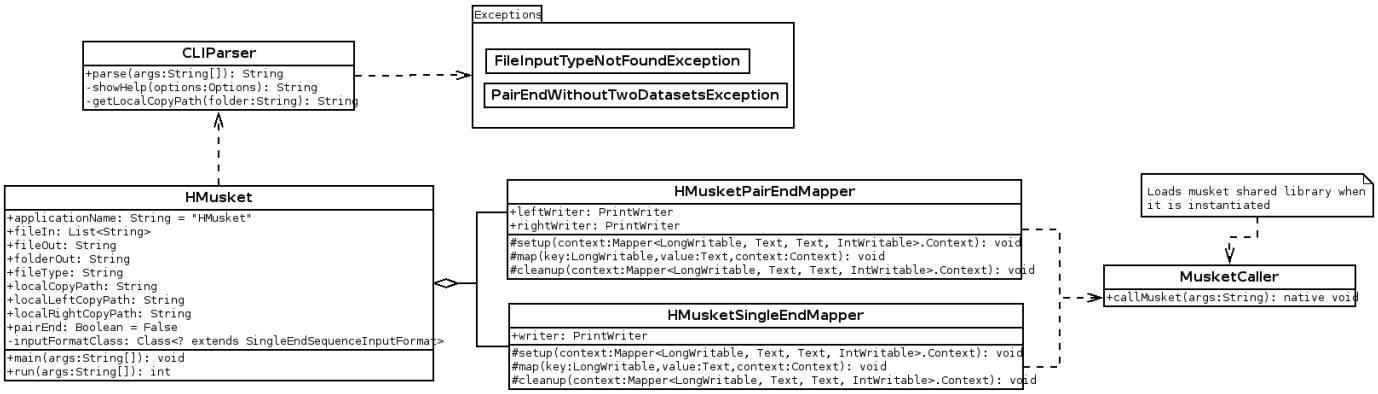

Figura 1: Diagrama de clases de HMusket

configurado en el entorno. Además de todo esto, BDEv provee de estadísticas e informes de evaluación acerca de cómo fue el rendimiento de la aplicación durante la ejecución. No obstante, el principal motivo para utilizar esta herramienta fue para poder ejecutar la aplicación distribuida en el sistema de colas del clúster, ya que actualmente no hay una solución unificada para la correcta integración este tipo de frameworks Big Data en dichos sistemas de colas.

Durante el análisis de Musket y HMusket se utilizaron dos conjuntos de datos genéticos de tipo Fastq que están disponibles públicamente: SRR921889 y SRR948355. El primero tiene formato single-end y consta de unas 50 millones de secuencias de 100 bases cada una. El segundo es un conjunto de datos paired-end (es decir, dos ficheros de entrada) con 69 millones de secuencias (cada uno) de 101 bases.

V-B. Análisis de Musket

Dado que Musket es un software acelerado mediante OpenMP (memoria compartida) el número máximo de threads con los que se pudo evaluar en el clúster fueron 16, y, teniendo en cuenta que se requieren por lo menos 2 threads para funcionar debido al patrón maestro/esclavo, se decidió evaluar el rendimiento de la aplicación con un número de hilos en potencias de 2, es decir: 2, 4, 8, 16.

Tal y como se puede apreciar en la Tabla II, la paralelización de Musket puede alcanzar una aceleración de aproximadamente 4.69 para el conjunto single-end y 8.64 para el paired-end. Resulta curioso que siendo el segundo conjunto más grande lo procese en menos tiempo. Esto puede ser debido que a los k-mers que conforman el espectro del conjunto paired-end son menores y por lo tanto se repiten más que en el primero, permitiendo un procesamiento más rápido del espectro. Respecto a las aceleraciones obtenidas en la evaluación experimental, se puede concluir que Musket escala razonablemente al aumentar el número de cores.

V-C. Análisis de HMusket

Para realizar las pruebas de HMusket se estudiaron y valoraron las posibles combinaciones de tareas Map a ejecutar por nodo que podían ser más eficientes. Debido a que cada nodo en Plutón se compone de 2 procesadores de 8 núcleos cada uno (véase Tabla I), se estableció como máxima repartir en cada nodo a lo sumo 1 mapper por cada procesador, es decir, 1 o 2 mappers por cada nodo de cómputo. Esto implica que se tenga que ajustar la memoria y el heap de los mappers: 50 Gb en caso de 1 mapper y 25 Gb en caso de 2 mappers por nodo, y en ambos casos se establece el 80 % de la memoria para el tamaño del heap. Por lo tanto, las combinaciones que se decidieron evaluar en los experimentos son los que se muestran en la Tabla III junto con los resultados obtenidos.

Tabla II: Resultados experimentales de Musket (en minutos)

Dataset	Threads	Tiempo
SRR921889	2	809.35
	4	524.28
	8	284.20
	16	172.28
SRR948355	2	810.03
	4	270.25
	8	144.56
	16	93.68

Durante la realización de los experimentos se realizaron 5 ejecuciones para cada prueba. El motivo de esto es minimizar el efecto que puede tener la variabilidad de tiempo entre diferentes ejecuciones en los resultados. Ya que Hadoop es un framework desarrollado en Java, y como todo software desarrollado con esta tecnología el acceso a los recursos proporcionados por la máquina virtual puede introducir ruido en la toma de tiempos (e.g. retardos en el acceso a discos u reserva de memoria). El tiempo de ejecución que se reporta en este articulo es la mediana de los 5 valores obtenidos. Se procedió de igual manera para los tiempos de la operación merge de todos los ficheros de salida.

Dados los resultados obtenidos en los experimentos realizados con HMusket se puede concluir que el procesamiento que realiza esta herramienta sobre los conjuntos de datos de entrada escala de una manera linear. Cabe destacar que se obtiene una aceleracion de 31.21, para el conjunto de datos single-end, en la aplicación Hadoop (configurada con dos mappers por nodo) frente a la versión de memoria compartida ejecutada con 16 threads. A su vez, para la corrección de

Tabla III: Resultados experimentales de HMusket (en minutos)

Dataset	Número de nodos	Mapper/Nodo	Threads/Nodo	Memoria/Mapper	Tiempo corrección	Tiempo merge
SRR921889	4	1	16	50 GB	21.70	2.08
	4	2	8	25 GB	10.17	1.36
	8	1	16	50 GB	9.44	2.11
	8	2	8	25 GB	5.52	1.56
SRR948355	4	1	16	50 GB	42.10	5.36
	4	2	8	25 GB	30.83	6.36
	8	1	16	50 GB	23.61	5.30
	8	2	8	25 GB	16.36	5.41

conjuntos de datos paired-end se obtiene una aceleración de 5,72. En ambos casos, la configuración que utiliza dos mappers por nodo resulta la más eficiente, teniendo que dividir la memoria disponible entre dos. Resulta interesante destacar las diferencias entre ejecutar dos mappers por nodo frente a ejecutar 1 mapper por nodo, para justificar que el uso de dos mappers por nodo es la configuración óptima. Por ejemplo, en el experimento donde se procesa el conjunto de datos en formato single-end con cuatro nodos, en el caso de procesar esta información con un solo mapper por nodo se obtienen 21,70 minutos. Por contra, si se procesa esta información con dos mappers por nodo se logra reducir este tiempo a 10,17 minutos. Es decir, con este cambio de configuración se obtienen una aceleración de 2,13.

Respecto a los tiempos obtenidos en la fase del merge de las salidas, se puede apreciar que apenas hay una gran variabilidad entre las distintas configuraciones. No obstante cabe destacar, como es lógico, que un mayor número de mappers implica un incremento del tiempo de merge. Pese a esto, hay que recordar que hablamos de conjuntos de datos del orden de 16 Gb (en el caso del single-end) y 44 Gb (22 Gb cada fichero en el caso del modo paired-end) y que la penalización es del orden de medio minuto.

En resumen, el uso de dos mappers por nodo, en las condiciones experimentales ya indicadas en la Tabla I, implica una aceleración de hasta 2,13 pese a penalizar con aproximadamente 30 segundos esta mejora en la fase de unión de todos los archivos de salida.

VI. Conclusiones y trabajo futuro

Teniendo en cuenta todo lo indicando a lo largo de este artículo se puede concluir que Musket es una herramienta muy potente y precisa, de hecho de las mejores que hay actualmente, pero su implementación en memoria compartida no puede hacer frente a la alta demanda de datos que ha surgido a lo largo de los últimos años.

Una buena alternativa para solucionar este problema es implementar o reutilizar dicho software en un entorno de memoria distribuida, como por ejemplo Hadoop, mediante el paradigma MapReduce, tal y como hace HMusket. De esta manera se pueden evitar cuellos de botella y preprocesados innecesarios de datos al inicio del análisis, obteniendo aceleraciones de hasta 31.21 además de una escalabilidad lineal en base al número de nodos. Esta herramienta se encuentra disponible en https://github.com/luislorenzom/hmusket.

Como trabajo futuro sería interesante realizar ciertas modificaciones tanto en la librería Musket como en la aplicación Hadoop para evitar las comunicaciones usando la entrada y salida estándar en lugar de ficheros y comparar el rendimiento de esta segunda versión frente a la aquí expuesta. Además, otro estudio que podría ser de gran interés sería analizar cómo escala el tiempo de la operación merge a medida que aumenta el tamaño del conjunto de datos de entrada y por consiguiente el número de mappers que utiliza la aplicación.

Referencias

[1] S. Behjati and P. S. Tarpey, "What is next generation sequencing?" *Arch Dis Child Educ Pract Ed*, vol. 98, no. 6, pp. 236–238, 2013.

[2] L. Garibyan and N. Avashia, "Polymerase chain reaction," *J. Invest. Dermatol.*, vol. 133, no. 3, pp. 1–4, 2013.

[3] M. Garcia-Diaz and K. Bebenek, "Multiple functions of DNA polymerases," *CRC Crit Rev Plant Sci*, vol. 26, no. 2, pp. 105–122, 2007.

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] The Apache software foundation [Internet], "Apache Hadoop," 2006 [cited 12 June 2018]. [Online]. Available: http://hadoop.apache.org

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*. Incline Village, NV, USA, 2010, pp. 1–10.

[7] Oracle corporation [Internet], "Java Native Interface," 2011 [cited 12 June 2018]. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[8] Aman Mangal, Chirag Jain, "Error correction in high-throughput short-read data on GPU (Mid-Term Progress)," 2014 [cited 17 June 2018]. [Online]. Available: https://github.com/mangalaman93/cude-ec

[9] Y. Liu, B. Schmidt, and D. L. Maskell, "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," *BMC Bioinformatics*, vol. 12, no. 1, p. 85, 2011.

[10] "SOAP - Short Oligonucleotide Analysis Package," 2007 [cited 17 June 2018]. [Online]. Available: http://soap.genomics.org.cn/soapdenovo.html

[11] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner, "Error correction of high-throughput sequencing datasets with non-uniform coverage," *Bioinformatics*, vol. 27, no. 13, pp. 137–141, 2011.

[12] K. Sameith, J. G. Roscito, and M. Hiller, "Iterative error correction of long sequencing reads maximizes accuracy and improves contig assembly," *Brief. Bioinformatics*, vol. 18, no. 1, pp. 1–8, 01 2017.

[13] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," *Genome Biol.*, vol. 11, no. 11, pp. 116 – 129, 2010.

[14] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner, "Error correction of high-throughput sequencing datasets with non-uniform coverage," *Bioinformatics*, vol. 27, no. 13, pp. 137–141, 2011.

[15] Y. Liu, J. Schröder, and B. Schmidt, "Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data," *Bioinformatics*, vol. 29, no. 3, pp. 308–315, 2013.

[16] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0," 2013 [cited 17 June 2018]. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[17] L. Ilie, F. Fazayeli, and S. Ilie, "HiTEC: accurate error correction in high-throughput sequencing data," *Bioinformatics*, vol. 27, no. 3, pp. 295–302, 2011.

[18] J. Schroder, H. Schroder, S. J. Puglisi, R. Sinha, and B. Schmidt, "SHREC: a short-read error correction method," *Bioinformatics*, vol. 25, no. 17, pp. 2157–2163, 2009.

[19] J. M. Mullaney, R. E. Mills, W. S. Pittard, and S. E. Devine, "Small insertions and deletions (INDELs) in human genomes," *Hum. Mol. Genet.*, vol. 19, no. R2, pp. 131–136, 2010.

[20] I. Akogwu, N. Wang, C. Zhang, and P. Gong, "A comparative study of k-spectrum-based error correction methods for next-generation sequencing data analysis," *Human Genomics*, vol. 10, no. 2, p. 20, 2016.

[21] C. C. Chen, Y. J. Chang, W. C. Chung, D. T. Lee, and J. M. Ho, "CloudRS: An error correction algorithm of high-throughput sequencing data based on scalable framework," in *Proceeding of the IEEE International Conference on Big Data, Santa Clara, CA, USA*, 2013, pp. 717–722.

[22] R. R. Expósito, L. L. Mosquera, and J. González-Domínguez, "Hadoop Sequence Parser (HSP) library," 2018 [cited 12 June 2018]. [Online]. Available: https://github.com/rreye/hsp

[23] "Pluton cluster's main page," 2013 [cited 17 June 2018]. [Online]. Available: http://pluton.des.udc.es/

[24] J. Veiga, J. Enes, R. R. Expósito, and J. Touriño, "BDEv 3.0: Energy efficiency and microarchitectural characterization of Big Data processing frameworks," 2018 [cited 17 June 2018]. [Online]. Available: http://bdev.des.udc.es/