UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS)

Tesis doctoral

BIG DATA MEETS HIGH PERFORMANCE COMPUTING: GENOMICS AND NATURAL LANGUAGE PROCESSING AS CASE STUDIES

Presentada por:

José Manuel Abuín Mosquera

Dirigida por:

Juan Carlos Pichel Campos
Tomás Fernández Pena

Santiago de Compostela, septiembre de 2017

Juan Carlos Pichel Campos, Profesor Contratado Doctor del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

Tomás Fernández Pena, Profesor Titular del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

HACEN CONSTAR:

Que la memoria titulada **Big Data meets High Performance Computing: Genomics and Natural Language Processing as case studies** ha sido realizada por **José Manuel Abuín Mosquera** bajo nuestra dirección en el Centro Singular de Investigación en Tecnoloxías da Información de la Universidade de Santiago de Compostela, y constituye la Tesis que presenta para optar al título de Doctor.

Santiago de Compostela, septiembre de 2017

Juan Carlos Pichel Campos
Director/a de la tesis

Tomás Fernández Pena
Director/a de la tesis

José Manuel Abuín Mosquera
Autor/a de la tesis

Juan Carlos Pichel Campos, Profesor Contratado Doctor del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

Tomás Fernández Pena, Profesor Titular del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

como Director/res de la tesis titulada:

Big Data meets High Performance Computing: Genomics and Natural Language Processing as case studies

Por la presente DECLARAN:

Que la tesis presentada por Don **José Manuel Abuín Mosquera** es idónea para ser presentada, de acuerdo con el artículo 41 del *Regulamento de Estudos de Doutoramento*, por la modalidad de compendio de ARTÍCULOS, en los que el doctorando ha tenido participación en el peso de la investigación y su contribución fue decisiva para llevar a cabo este trabajo. Y que está en conocimiento de los coautores, tanto doctores como no doctores, participantes en los artículos, que ninguno de los trabajos reunidos en esta tesis serán presentados por ninguno de ellos en otras tesis de Doctorado, lo que firmo bajo mi responsabilidad.

Santiago de Compostela, septiembre de 2017

Juan Carlos Pichel Campos
Director/a de la tesis

Tomás Fernández Pena
Director/a de la tesis

A Martín e a Verónica
Por cambiarme a vida. De boa, a mellor

The two most important days in your life are the day you are born and the day you find out why.

Mark Twain

There are things known and there are things unknown, and in between are The Doors

Jim Morrison

Agradecementos

Non podo comezar esta sección doutra maneira que non sexa agradecendo a Pichel e a Tomás por darme esta oportunidade. Primeiro co proxecto de PLN e despois coa bolsa FPI. Sen o seu apoio dende un principio esta tese non sería posible. Ademais diso nunca deixaron de recibirme amablemente sempre que petaba na súa porta cun novo problema (ou non tan novo).

Quero tamén agradecer ó resto do Grupo de Arquitectura de Computadores da USC, en especial a Fran, Caba e Dora. Ós primeiros porque a miña andanza no mundo "paralelo" empezou con eles durante o meu TFM, e a Dora porque dela aprendín moitísimo no referente a docencia e á vida universitaria como investigador. Así mesmo, tamén darlle as gracias a Pablo Gamallo e a Jorge Amigo, pola súa axuda e sempre boa disposición coas miñas preguntas no tocante ás súas especialidades, PLN e Bioinformática respectivamente.

Non podo esquecerme do resto de compañeiros na miña vida profesional. Comezando, primeiro, polo Cesga, onde me deron a miña primeira oportunidade. Alí tiven a sorte de coñecer e aprender de grandes profesionais e amigos. María José, Manuel Gromaz, Silvia, Cecilia e Javi. Despois, co salto ó Departamento de Matemática Aplicada da USC, tiven o luxo de traballar con Alfredo Bermúdez, Julio, Chuco, Ángel e Víctor, dos cales aprendín tanto tecnicamente como coma persoa, e cos cales gardo grandes recordos.

E no tocante ó CiTIUS, pouco que dicir, e ó mesmo tempo, moito. Esta tese non tería visto o fin sen as longas conversas, apoios, comidas e cafés compartidos con Esteban, Bea, Julio, Diego Rodríguez, Guillermo, Vanesa, Rodrigo e Carlos Bran.

I would also like to thank the High Performance Computing Research Group at the Johannes Gutenberg Univertisät for the great experience in the three and a half months I spent there. I learned a lot from them, specially from Bertil Schmidt, Tu Tuan Tran, Christian Hundt, André Müller and André Weißenberger.

Tamén quero darlle as gracias, como non, á miña familia, por estar sempre presente durante o desenvolvemento deste traballo. Ademais, quero agradecer ás comunidades de StackOverflow, Wikipedia e GitHub, así coma a Linus Torvalds por Linux e a Dennis Ritchie pola linguaxe C e por Unix.

Para rematar, quero agradecer á Xunta de Galicia pola súa financiación no proxecto "Computación de Altas Prestacións para o Procesamento da Linguaxe Natural" (EM2013/041), coa cal estiven contratado, e ó Ministerio de Economía y Competitividad pola financiación do proxecto "Soluciones Hardware y Software para la Computación de Altas Prestaciones" (TIN2013-41129-P) e a correspondente beca FPI (BES-2014-067914). Tamén, como non, agradecer ó Cesga e a Amazon AWS pola posibilidade de empregar as súas instalacións.

Santiago de Compostela, septiembre de 2017

Resumen

En los últimos años las tecnologías Big Data han tenido un auge enorme dentro del mundo de la industria y la investigación. Esto se ha debido, principalmente, a su capacidad para realizar el procesamiento de grandes volúmenes de datos usando arquitecturas paralelas de un modo sencillo, eficiente y completamente transparente para el usuario. Dicho de otro modo, las tecnologías Big Data han acercado la programación paralela clásica en memoria distribuida a un público más general, facilitando en gran medida su adopción sin una pérdida importante de rendimiento, y ese es parte de su éxito.

Dicho éxito se manifiesta en su uso en centros de investigación punteros a nivel mundial. Por ejemplo en el CERN, se usan tecnologías Big Data para procesar los datos producidos por el Gran Colisionador de Hadrones (LHC), o la NASA, que utiliza este tipo de tecnologías para tratar datos recibidos por grandes telescopios instalados en diferentes localizaciones a lo largo del mundo.

El origen de estas tecnologías puede encontrarse en el año 2004, cuando Google publica un trabajo donde presenta el modelo de programación MapReduce. A partir de ahí, y con el surgimiento de Apache Hadoop como implementación *Open Source* de dicho modelo, el crecimiento de este tipo de tecnologías ha sido exponencial.

Por otro lado, en el ámbito de la Computación de Altas Prestaciones (High Performance Computing o HPC), existe una carrera entre empresas, instituciones y centros de investigación para entrar en la era de los supercomputadores exascale. Estos sistemas de computación deberán ser capaces de realizar 10^{18} operaciones en punto flotante por segundo, es decir, tener un rendimiento de 1 EXAFLOP. Hoy en día aún estamos lejos de alcanzar dicha meta, ya que el supercomputador que se alza con el primer puesto en el TOP500 en Junio de 2017 es capaz de alcanzar los nada despreciables 125,4 PETAFLOPS $(125,4 \times 10^{15} FLOPS)$, pero todavía falta un orden de magnitud para alcanzar el EXAFLOP, lo que supone un enorme salto

tecnológico.

Para llegar a alcanzar el EXAFLOP de rendimiento, los supercomputadores necesitarán que el envío de datos se realice de un modo rápido y eficiente, tanto dentro de un mismo nodo como entre nodos diferentes. Esta es una tarea difícil de lograr en los grandes supercomputadores, así como en los programas con una alta demanda computacional, como los provenientes de problemas científicos y de análisis de datos. Además, las *Application Programming Interfaces* (APIs) deberán proveer al programador de métodos que le permitan llevar a cabo la explotación de cantidades excepcionales de paralelismo y, al mismo tiempo, hacerlo de modo que la facilidad de uso y la programabilidad no sea un problema, así como soportar arquitecturas heterogéneas, tales como las que incorporan GPGPUs o sistemas manycore. Otro requisito es el de dar soporte a mecanismos de tolerancia a fallos, mediante los cuales una aplicación se podría recuperar de un fallo software o hardware, para continuar con la ejecución normal del proceso justo desde el punto donde se produjo dicho error.

Las APIs de los lenguajes de programación de computación paralela clásicos (como por ejemplo MPI, OpenMP, etc) se encuentran en una etapa de desarrollo y mejora, con el objetivo de alcanzar los requisitos mencionados anteriormente. Por otro lado, los entornos de desarrollo Big Data (por ejemplo, Spark o Hadoop) ya cumplen con algunas de estas características, como la tolerancia a fallos o la facilidad de programación. Aún así, todavía no está claro que paradigma encaja mejor para alcanzar un alto rendimiento y, al mismo tiempo, manejar grandes cantidades de datos de una forma eficiente en un amplio rango de códigos científicos.

Además de lo expuesto en el párrafo anterior, existe una cierta tensión entre la necesidad de reducir el movimiento de datos y el potencial de organizar y ejecutar tareas de un modo dinámico (con el movimiento de datos que ello implica). El rol del usuario al tratar de balancear dichos parámetros es todavía un punto de debate. Aún así, podríamos preguntarnos, ¿existe una diferencia fundamental entre HPC y Big Data?, ¿o la diferencia simplemente reside en las aplicaciones y el software empleado? Mientras que la Computación de Altas Prestaciones se centra más en grandes cargas computacionales, las tecnologías Big Data tienen como objetivo aplicaciones que necesitan manejar grandes y complejos conjuntos de datos. Dichos conjuntos de datos son, habitualmente, del orden de varios PebiBytes o TebiBytes de tamaño.

A primera vista, estas diferencias entre Big Data y HPC pueden resultar extrañas, ya que, en el fondo, ambos ecosistemas engloban tecnologías que permiten realizar tareas de cómputo en paralelo con la consiguiente reducción de tiempos de ejecución y mejora del rendimiento.

Entonces, ¿a qué se deben estas diferencias? Podemos destacar varios factores. El primero es que, con el surgimiento de las tecnologías Big Data, también ha surgido todo un ecosistema de nuevas aplicaciones asociadas a ellas (schedulers, sistemas de fichero paralelos, etc). Estas aplicaciones ya tenían anteriormente sus equivalentes en el mundo HPC. La gran diferencia es que las nuevas tecnologías incorporan funcionalidades que son necesarias para implementar algunas de las características típicas de las aplicaciones Big Data (por ejemplo, la tolerancia a fallos). La consecuencia más inmediata y evidente de esta divergencia es la existencia de dos ecosistemas paralelos y completamente diferentes e incompatibles (al menos de momento) entre el mundo HPC y en el mundo Big Data.

Otro de los factores causantes de dicha divergencia es, aunque pueda parecer poco importante en primera instancia, el lenguaje de programación empleado. Normalmente los lenguajes de programación con más aceptación en el mundo HPC son C, C++ o Fortran. Mientras, las tecnologías Big Data suelen utilizar lenguajes de más alto nivel, como por ejemplo, Java, Python o Scala, ya que estos lenguajes ofrecen una mejor programabilidad. Esto resulta ser un problema, ya que las aplicaciones o librerías tradicionalmente empleadas en HPC, o incluso en otras áreas científicas, no están implementadas mediante los lenguajes que se están usando en Big Data. Las tecnologías Big Data se han orientado, desde su inicio, hacia su uso por parte de los denominados "científicos de datos", más preocupados por el tratamiento estadístico de los datos que por las características a bajo nivel de sus aplicaciones. Por lo tanto, tienden a usar lenguajes de más alto nivel como los citados anteriormente, o incluso de propósito específico, como por ejemplo SQL o R, para desarrollar códigos de una forma rápida.

Estos dos factores son, sin duda, una dificultad en el camino hacia los supercomputadores exascale. Varias investigaciones en el área del HPC ya han manifestado la conveniencia de que los dos mundos, HPC y Big Data, han de converger para poder alcanzar este objetivo. De momento, sin embargo, no se han propuesto metodologías o tecnologías que hagan que los dos mundos coexistan o converjan.

A medida que las investigaciones científicas demanden una mayor velocidad de cómputo y capacidad para analizar datos, la potencial interoperabilidad de estos dos mundos es crucial para el futuro. En esta tesis, se emplean tecnologías Big Data para tratar problemas científicos que son computacionalmente intensivos en cuanto a tiempo de ejecución (típico en problemas HPC) y, al mismo tiempo, tienen un gran tamaño en cuanto a datos de entrada (típico en problemas Big Data), con el objetivo de mejorar el tiempo de ejecución, la escalabilidad y la eficiencia.

Dichos problemas científicos abordados en esta tesis tienen una serie de características comunes que los hacen adecuados para demostrar los beneficios generados por la sinergia entre los mundos HPC y Big Data:

- Requerir una gran cantidad de datos de entrada.
- Tener una elevada carga computacional, ya que las aplicaciones HPC son computacionalmente intensivas.
- Deben generar una gran cantidad de datos de salida. En muchos casos, aplicaciones de ambos mundos forman parte de pipelines de trabajo que proporcionan datos de entrada para otras aplicaciones. Por lo tanto, la salida de dichas aplicaciones también suele generar una gran cantidad de información.
- Deben estar escritos, al menos en su mayor parte, en un lenguaje de programación típico de las aplicaciones HPC.

Teniendo en cuenta dichos requisitos, los problemas científicos considerados en esta tesis se engloban dentro de las áreas científicas de la **Genómica** y del **Procesamiento del Lenguage Natural (PLN)**.

El área del **Procesamiento del Lenguaje Natural (PLN)**, está considerada como una de las metodologías más apropiadas para poder estructurar y organizar la información textual accesible a través de Internet. El procesamiento lingüístico de grandes cantidades de texto es una tarea compleja que requiere del uso de varias subtareas organizadas en módulos interrelacionados. Estos módulos son necesarios para poder llevar a cabo tareas más complejas, como la traducción automática, la recuperación de información o sistemas de vigilancia tecnológica. Generalmente, se necesita que el procesamiento lingüístico en tareas NLP sea lo más precisa y eficiente posible.

Uno de los mayores problemas que presentan los módulos PLN es su alto coste computacional y sus dificultades de escalabilidad. Esto los hace inviables para el análisis de grandes volúmenes de documentos (GibiBytes e incluso TebiBytes). De este modo, en esta tesis hemos considerado que el uso de soluciones HPC y Big Data se hace indispensable si se quiere reducir de forma notable los tiempos de cómputo, mejorar la escalabilidad del sistema y abordar problemas de un tamaño aún mayor. El conjunto de módulos PLN que se han seleccionado han sido los que permiten realizar la identificación y clasificación de entidades con nombre o NERC por sus siglas en inglés *Named Entity Recognition and Classification*.

El NERC es una línea de investigación en sí misma dentro del Procesamiento de Lenguaje Natural. El objetivo de esta línea es reconocer, identificar y clasificar nombres dentro de un texto. Este proceso se lleva a cabo con la ayuda de una lista predeterminada de categorías, por ejemplo, Persona, Lugar, Organización, etc. Las herramientas del estado del arte en esta área usan tanto técnicas lingüísticas basadas en gramáticas como modelos estadísticos. Normalmente los sistemas basados en gramáticas hechas a mano obtienen mejor precisión, pero con el coste de meses de trabajo de expertos lingüistas con experiencia en computación. Por otro lado, los sistemas NERC estadísticos requieren de una gran cantidad de datos de entrenamiento anotados a mano. También existen los enfoques semi-supervisados, que han sido sugeridos para evitar parte del esfuerzo de anotación.

Todos estos enfoques adolecen de la misma característica, que es la capacidad de procesar grandes cantidades de datos en un tiempo razonable. Por ejemplo, procesar toda la Wikipedia en español conlleva un tiempo de ejecución de alrededor de 19 días con un sistema NERC del estado del arte incluido en el repositorio Linguakit[1]. Los módulos PLN de dicho sistema están implementados en lenguaje Perl, y funcionan como un pipeline de trabajo.

El lenguaje Perl se usa frecuentemente en tareas de PLN, ya que está preparado para tratar con datos de tipo texto, debido sobre todo a su potencia en el uso de expresiones regulares. Sin embargo, el caso de emplear herramientas escritas en lenguaje Perl o en cualquier otro lenguaje diferente de Java con Apache Hadoop no está contemplado. Para poder realizar dicha tarea, Apache Hadoop permite emplear la herramienta Hadoop Streaming. Dicha herramienta permite al usuario ejecutar programas que siguen el modelo MapReduce empleando cualquier lenguaje de programación, con la única limitación de que el código debe leer los datos de entrada desde la entrada estándar, o *stdin*, y escribir los datos de salida en la salida estándar, o *stdout*. El problema es que dicha herramienta ha demostrado carecer de la eficiencia esperada. Debido a ello en esta tesis se han explorado otro tipo de soluciones a este problema, el cual también podría presentarse en otro tipo de aplicaciones.

Tras explorar dichas soluciones, se ha creado la herramienta **Perldoop**, que permite traducir códigos escritos en lenguaje Perl a códigos escritos en Java preparados para su ejecución con Apache Hadoop sin necesidad del módulo de Hadoop Streaming. Esto permite que investigadores en PLN puedan traducir y ejecutar sus códigos desarrollados en Perl en un entorno Big Data, con la consiguiente mejor eficiencia posible, escalando adecuadamente y, además, con la tolerancia a fallos que proporciona.

[1] https://linguakit.com

El proceso de traducción es relativamente sencillo, ya que tan sólo requiere por parte del usuario añadir una serie de *tags* o etiquetas en el código Perl y emplear unos *templates* o plantillas en lenguaje Java que ya provee la propia herramienta Perldoop. De este modo, los investigadores en el área del PLN no tienen que profundizar en un nuevo lenguaje o unas nuevas tecnologías con las que no tienen por qué estar familiarizados. Todo este proceso de traducción, así como de la implementación y resultados obtenidos, se tratan con profundidad en el Capítulo 2. Así mismo, también se muestran ejemplos de traducción sencillos, de modo que el usuario pueda comprender fácilmente el proceso.

Para obtener resultados de un problema real y con un software del estado del arte, en dicho capítulo se ha empleado Perldoop con los modulos NERC de Linguakit. La plataforma seleccionada para llevar a cabo los experimentos ha sido un clúster Big Data del Centro de Supercomputación de Galicia. Con estas características, los resultados han mostrado un speed-up de 57.9× usando 64 cores para los módulos NERC traducidos, frente al speed-up de 29.4× obtenido por Hadoop Streaming. Como se puede apreciar, el rendimiento de los módulos usando Perldoop es aproximadamente el doble que usando la herramienta Hadoop Streaming.

La segunda área científica que tratamos en esta tesis es la **Genómica**. El primer paso en un análisis genómico de ADN o ARN es siempre la lectura de una muestra biológica, para poder, de este modo, trasladar la información contenida en dicha muestra desde el entorno biológico al computador, y así poder analizar dichos datos computacionalmente. Esto se consigue mediante el uso de las tecnologías de ultrasecuenciación. Gracias a estas tecnologías, se han desarrollado máquinas que permiten introducir la muestra y obtener directamente los datos en un formato digital, es decir, después del proceso de ultrasecuenciación, los datos tienen forma de uno o varios ficheros de texto que siguen un determinado formato.

En los últimos años, dichas tecnologías de ultrasecuenciación han dado un salto importantísimo en lo referente a la cantidad de datos que se pueden obtener de muestras, así como en la velocidad a la que se pueden obtener dichos datos. Dicho salto en lo referente a estas tecnologías es en parte debido a los avances científicos llevados a cabo por compañías como, por ejemplo, Illumina, o a centros como el Wellcome Trust Sanger Institute del Reino Unido.

Debido a la velocidad en la obtención de datos y a la cantidad obtenida de los mismos, éstos pueden alcanzar fácilmente el orden de varios GibiBytes o TebiBytes de datos, dependiendo del tipo de muestra. El problema que se presenta ahora es dar sentido científico a dichos datos, ya que la velocidad a la que aumenta su tamaño está sobrepasando a la velocidad a la que crece la capacidad de cómputo de un procesador.

1	2	3	4	5	6
AACGT- ACCGTT	-AACGT ACCGTT	A-ACGT ACCGTT	AACGT—— ——ACCGTT	AA-CGT- A-CCGTT	-A-A-C-G-T- A-C-C-G-T-T

Tabla 1: Seis posible alineamientos para las secuencias de ejemplo.

Normalmente, los profesionales en el área emplean flujos de trabajo descritos en las *GATK Best Practices* del Broad Institute, del MIT y Harvard para dar sentido científico, o analizar, los datos obtenidos. Dichas buenas prácticas describen los pasos a seguir, así como el software a utilizar en cada uno de esos pasos, para elaborar una serie de análisis concretos. Algunos de estos pasos son comunes a varios de los flujos de trabajo.

Una de las etapas que son comunes a varios de estos flujos de trabajo es la del alineamiento o mapeado de secuencias. Para explicar lo que es el alineamiento, primero hay que tener en cuenta que las secuencias de ADN o proteínas pueden cambiar su codificación mientras evolucionan en el tiempo. Los tipos más simples de mutaciones son mutaciones puntuales e inserciones/extracciones, también conocidos como *indels* (insertion/deletion). El alineamiento trata de identificar estas mutaciones mediante algoritmos de alineamiento clásicos. Por ejemplo, al alinear las dos secuencias de ADN, *AACGT* y *ACCGTT*, algunos de los posibles resultados son los que se pueden ver en la Tabla 1, donde cada carácter representa una base nitrogenada y el guión representa un hueco.

La cuestión ahora sería saber cuál de dichos alineamientos es el "mejor". Para ello, existen varios algoritmos de puntuación o *score* explicados en la literatura. Las referencias a la literatura donde se explican dichos algoritmos de puntuación se indican en la Introducción de este documento. Sin embargo, y teniendo en cuenta que hay varios posibles alineamientos y que es necesario puntuarlos, y que al mismo tiempo estamos hablando de una gran cantidad de datos, este paso es uno de los más costosos computacionalmente dentro de los flujos de trabajo mencionados. Al mismo tiempo, es uno de los pasos más fundamentales, por lo tanto, obtener herramientas que permitan realizar este alineamiento de una forma eficiente y escalable es un requisito indispensable.

Dentro de las herramientas del estado del arte, existen varias que permiten llevar a cabo el alineamiento de secuencias. En concreto, en el caso de alineamiento de secuencias cortas (de menos de 100 caracteres o pares de bases), la herramienta BWA (Burrows-Wheeler Aligner) es una de las más utilizadas, como así lo refleja el hecho de que es la indicada en las *GATK Best Practices* para realizar la fase de alineamiento. Dicha herramienta está implementada

en C y utiliza paralelismo a nivel de hilo (es decir, en memoria compartida). Debido a su amplia aceptación entre la comunidad de investigadores en Bioinformática y a los detalles de su implementación, se ha seleccionado esta herramienta como caso de estudio en esta tesis. A pesar de disponer de una versión paralela para sistemas de memoria compartida, el alineamiento con BWA puede suponer varios días de cómputo dependiendo del tamaño de los datos de entrada.

Debido a lo expuesto en el párrafo anterior, y al tratar con una gran cantidad de datos de entrada proveniente de las máquinas de ultrasecuenciación, las tecnologías Big Data parecen las adecuadas para tratar con este problema. Al mismo tiempo, al tratarse de un problema computacionalmente intensivo, las tecnologías HPC también podrían ser adecuadas. Por ello, se ha implementado la herramienta **BigBWA**, la cual emplea Hadoop como tecnología Big Data para poder realizar en paralelo el alineamiento, mientras que internamente emplea los algoritmos implementados en BWA. Dichos algoritmos están escritos en lenguaje C, y se hace uso de ellos mediante llamadas desde Java en las partes que son computacionalmente intensivas mediante el uso de JNI (*Java Native Interface*). Debido a que las secuencias de entrada son totalmente independientes entre sí, estamos hablando de un problema de los denominados *Embarrassingly Parallel*, ya que los datos se dividen en fragmentos y cada uno de dichos fragmentos pueden ser procesados independientemente de los demás.

Siguiendo esta estrategia, cuyos detalles se pueden ver en el Capítulo 3, y empleando un clúster Big Data desplegado en Amazon Web Services se han llevado a cabo una serie de experimentos que demuestran la viabilidad de esta herramienta. Los datos de entrada se han tomado de un repositorio público de secuencias de ADN humano, que proviene del proyecto *1000 Genomes*. Los resultados de dichos experimentos han mostrado como BigBWA consigue una aceleración de 36.6× empleando 64 cores con respecto a la versión secuencial de BWA (1 thread), mientras que, al mismo tiempo, es más rápido que otras herramientas similares del estado del arte. Además incorpora características típicas de los ecosistemas Big Data, como la tolerancia a fallos.

Dentro del mundo de las tecnologías Big Data, los avances se producen muy rápidamente. Así lo demuestra la aparición de Apache Spark, un motor que mejora Apache Hadoop en varios aspectos. Por ejemplo, permite ir más allá del modelo MapReduce o proporciona mejoras en el acceso a disco. Debido a ello, en esta tesis se ha implementado también una versión de BWA empleando Apache Spark, surgiendo así **SparkBWA**.

SparkBWA sigue la misma estrategia que su predecesor, BigBWA. Es decir, realiza la di-

visión de datos y tareas empleando tecnologías Big Data, en este caso Apache Spark, mientras que el cómputo intensivo requerido por el algoritmo de alineamiento se realiza con el código en C de BWA mediante JNI. De este modo, las ventajas obtenidas en BigBWA se mantienen y, al mismo tiempo, se incluyen mejoras que proporciona Apache Spark. Por poner un ejemplo, una fase de preprocesado de datos que requería BigBWA, ahora con SparkBWA se realiza de forma más eficiente empleando funciones nativas de Spark.

Todos los detalles de la implementación, así como las mejoras mencionadas en el párrafo anterior, se pueden ver en el Capítulo 4. En dicho capítulo también se presentan los resultados obtenidos. SparkBWA consigue una aceleración de hasta 85.6× usando 128 cores con respecto a la versión secuencial de BWA (1 thread), mientras que BigBWA y otras herramientas del estado del arte se quedan alrededor de 66× con el mismo número de cores y también con respecto a la versión secuencial de BWA (1 thread).

Uno de los objetivos de la comparación de secuencias de proteínas es descubrir similaridades estructurales o funcionales entre proteínas. Biológicamente, proteínas similares podrían no mostrar una similaridad clara. Por ejemplo, si la similitud de las secuencias es baja, el alineamiento de un par de secuencias puede fallar al identificar secuencias biológicamente relacionadas. Sin embargo, la comparación simultánea de varias secuencias a menudo permite encontrar similitudes que son imposibles de identificar en el caso de un par de secuencias. Dicho es lo que se conoce como *Multiple Sequence Alignment* o MSA, en el cual se pretende alinear varias secuencias al mismo tiempo. La mayor parte de implementaciones de algoritmos que llevan a cabo MSA llevan a problemas de optimización de combinatoria NP completos. Como parte de esta tesis también se ha llevado a cabo un trabajo en el que se integra un software para MSA, denominado PASTA, en un entorno Big Data, surgiendo así la herramienta **PASTASpark**.

En este caso, PASTA engloba diversas herramientas para llevar a cabo el alineamiento múltiple. Dichas herramientas están implementadas en diferentes lenguajes de programación, como por ejemplo, Python, e incluso se realizan llamadas a binarios ejecutables, por lo que es otro candidato ideal para este nuestros objetivos en esta tesis. Cada una de dichas herramientas se utiliza en diversas fases. En este caso, el trabajo se ha centrado en paralelizar una fase en la que interviene el alineador conocido como MAFFT.

PASTA incluye por defecto un modo paralelo, en el que se ejecutan varios procesos al mismo tiempo, con la característica de que dichos procesos sólo pueden ser ejecutados en la misma máquina. Sin embargo, con PASTASpark, la fase donde interviene MAFFT, que es la

más costosa computacionalmente, se ejecuta en un cluster de computación empleando Apache Spark. Los detalles de nuestra propuesta se muestran en el Capítulo 5. De todos modos, cabe mencionar que en este caso una de las fases de PASTA supone un cuello de botella que no permite a PASTASpark escalar de manera adecuada. Debido a eso, los resultados se ven limitados por la ley de Amdahl, aunque están muy próximos a dicho límite. Por ejemplo, empleando un cluster en Amazon Web Services, con el conjuntos de datos de entrada más grande del que se dispone, se baja de las 24 horas de ejecución, mientras que usando la versión original de PASTA se necesitan varios días y no se consigue completar la ejecución.

Para finalizar este resumen, mencionar que las conclusiones globales obtenidas de todos los trabajos aquí expuestos se encuentran detalladas en el Capítulo 6.

Contents

CHAPTER 1

INTRODUCTION

1.1. Motivation

The human being has entered in the era of data. According to IBM, every day 2.5 quintillion bytes of data are created, in such a way that 90% of the data all over the world were created just only in the last two years [1]. This data comes from everywhere: sensors that are dedicated to gather climate information, posts to social networks, digital pictures and videos, smartphones GPS signals, etc. But the Internet of Things or the Social Networks are not the only responsible of this data explosion, science also plays an important role. As an example, the Large Hadron Collider (LHC) in Geneve (Switzerland), generates about 30 PetaBytes of data per year [2], and in its first phase, the Square Kilometre Array (SKA) radio telescope will produce 160 TeraBytes of raw data per second [3]. All this data is what is known as **Big Data**.

The task of processing and generating knowledge from this huge amount of information can be an unaffordable problem. We can summarize the main challenges to deal with Big Data in the following way:

- Data is usually not structured.
- The computing time to process and access this information is typically very high.
- The memory consumed by scientific applications that make use of this data exceeds by far the memory that is typically installed on a workstation.

Name	Description	Performance
`Sunway TaihuLight`	National Supercomputing Center in Wuxi (China). Supercomputer at the top of the TOP500 list (June 2017).	125.4 PFLOPS ($\times 10^{15} FLOPS$) [7]
`FinisTerrae II`	Galicia Supercomputing Centre (CESGA) cluster.	328 TFLOPS ($\times 10^{12} FLOPS$) [8]
`PlayStation 4`	Videogame console by Sony.	1.8 TFLOPS ($\times 10^{12} FLOPS$) [9]
`Nexus 5`	Android mobile phone.	18 GFLOPS ($\times 10^{9} FLOPS$) [9]

Table 1.1: Comparing processing power of different platforms.

To overcome these limitations, Google developed the **MapReduce** model [4, 5]. MapReduce is a programming model that comes with an associated implementation for processing and generating big data sets on commodity clusters. In this model, the runtime takes care of:

- Partitioning the input data.
- Scheduling the program's execution across a set of machines.
- Handling machine failures.
- Managing inter-machine communications.

Since the release of the MapReduce model and the birth of its *Open Source* implementation, Apache Hadoop [6], the growth of Big Data technologies has been exponential.

On the other hand, nowadays, the High Performance Computing (HPC) community is involved in a race between companies, institutions, and research centres to reach the Exascale milestone. Exascale refers to supercomputers capable of executing 10^{18} floating point operations per second, this is, one EXAFLOP per second. To give some perspective to this number, some comparison data is displayed in Table 1.1. As the numbers reveal, there is still an order of magnitude to reach Exascale for the first supercomputer at the TOP500 [7] list.

To reach that performance, future supercomputers will need that data delivery to be fast and efficient, both from memory and disk, and also across the network and between processors. This is a difficult task to achieve in big supercomputers, and also in large computations, like those present in scientific and data analytics problems. Also, Exascale Application Programming Interfaces (APIs) will need to make easy for the programmer the exploitation

of exceptional amounts of parallelism in applications, enabling the processing of significant amounts of data, and supporting several different architectures, including those based upon heterogeneous cores or accelerators. These APIs and their implementations will need to carefully manage different kinds of memories within each node. Moreover, the need to conserve energy has led to an increased focus on reducing data motion at all levels of the memory hierarchy, from low cache levels to main memory, requiring a rethinking of algorithms as well as of the entire HPC software stack. In addition, Exascale execution software systems will need to ensure that jobs continue to run despite the occurrence of system failures and other kinds of hardware or software errors.

Traditional programming interfaces for expressing parallelism (in particular, MPI [10], OpenMP [11] and OpenACC [12]) are being further developed to achieve some of these requirements. On the other hand, Big Data frameworks (such as Apache Spark [13] or Apache Hadoop [6]) have already implemented fault tolerance and programmability requirements. However, it is still unclear which paradigm is a natural fit for expressing computations and handling data in a broad range of scientific application codes.

At the same time, there is a tension between the need to reduce data motion and the potential to dynamically schedule and execute tasks. The role of the user in balancing these is also still being debated. Yet, is there a fundamental difference between HPC and Big Data or does the difference only reside in the application and software usage? While HPC mostly focuses on large computational loads, Big Data targets applications that need to handle very large and complex data sets. These data sets are typically of the order of multiple terabytes or exabytes in size. Big Data applications are thus very demanding in terms of storage, to accommodate such a massive amount of data, while HPC is usually thought more in terms of pure computational needs [14].

At this point, and after this explanation, a comparison between the Big Data and HPC stacks is needed. A stack based on [15] can be seen in Figure 1.1. In this figure, we can observe how both ecosystems share some attributes, for example, a notable reliance on open source software and the x86 hardware. However, as scientific research increasingly depends on both high-speed computing and data analytics, the potential interoperability of these two ecosystems is crucial for the future. Also, despite the similarities, they differ markedly in their foci and technical approaches.

Regarding the HPC ecosystem, commodity clusters (Intel/AMD) and purpose built processors (IBM's BlueGene) that dominated the previous decade, have been augmented with

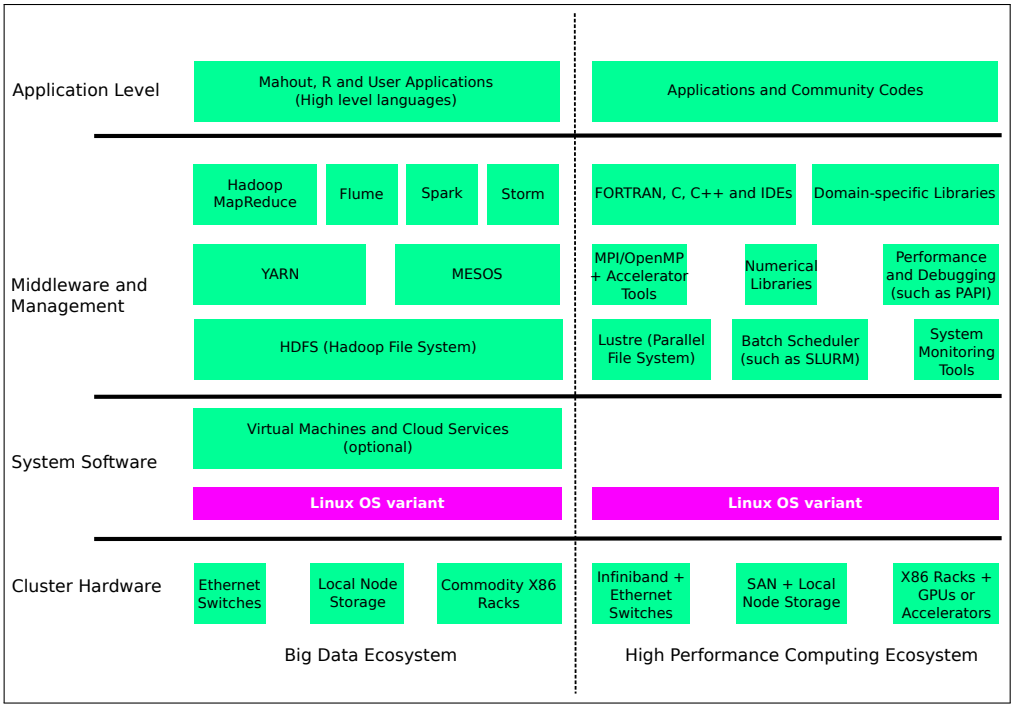


Figure 1.1: Stack comparison between Big Data and HPC ecosystems.

computational accelerators in the form of coprocessors, graphical processing units (GPUs) or FPGAs. They also include high-speed low-latency networks (such as Infiniband) and Storage Area Networks (SANs). This hardware ecosystem is optimized for performance, rather than for minimal cost or energy consumption.

On top of the hardware, Linux provides the operating system, augmented with parallel file systems (Lustre [16]) and batch schedulers (Torque [17] or SLURM [18]) for parallel job management. MPI [10] and OpenMP [11] are used for internode and intranode parallelism, augmented with CUDA [19] or OpenCL [20]. Numerical and domain specific libraries such as LAPACK [21] and PETSc [22] complete the software stack. Applications are typically written in Fortran, C or C++.

On the other hand, in the Big Data Ecosystem, clusters are typically based on Ethernet networks and local storage, which focus in cost and capacity. In this case, on top of the Linux operating system, HDFS [23] is commonly used, completed with YARN [24] or Mesos [25]

for job scheduling. In this case, on top of the schedulers, different technologies can be found, such as Spark or Hadoop MapReduce. Spark includes some libraries for Machine Learning or Graphs Processing, with some aspects related to numerical computation but more oriented to statistics than matrix algebra or simulation. Finally, on top of the Big Data stack, user applications are usually written in high level languages (Perl or Python, for example). This is another difference, since HPC programs are written in low level languages, which are better suited for performance optimization.

Despite the fact that both ecosystems share some ideas, philosophy, and even technology, they are very different. In addition, the barrier between Big Data technologies and classic High Performance Computing applications is still not clear. In this thesis, we use Big Data technologies to deal with some scientific problems that are computationally intensive regarding execution time (typical in HPC problems) and, at the same time, have a large input data size (typical in Big Data), with the objective of improving their execution time, scalability and efficiency. Conclusions derived from this work can clarify where this barrier stands, or even prove if this barrier exists at all, and maybe help to solve the key question that in recent times has arisen among the HPC community: should Big Data be considered part of the High Performance Computing field?

1.2. The MapReduce programming model: Apache Hadoop

Next, a brief overview of how the MapReduce programming model and Apache Hadoop work is introduced. In the MapReduce model the computation takes a set of *input key/value pairs*, and produces a set of *output key/value pairs*. The model expresses the computation as two functions written by the user: **map** and **reduce**.

The **Map** function takes as input a key/value pair and produces an intermediate list of key/value pairs. The function is called one time per each input key/value pair in the input data. Then, the intermediate values associated with the same key are grouped together. Sometimes, data need to be redistributed according to the intermediate keys, in a way that all the values belonging to the same key are in the same computing node. This is what is known as the *shuffle* phase. After that, data reaches the reduce function.

The **reduce** function, accepts an intermediate key and a list of values for that key. It operates with these values to conform a set of output data. The intermediate values are supplied to the user's reduce function via an iterator. This allows the user to handle lists of values that

are too large to fit in memory.

By using these two functions from the model, the user should be able to process this large amount of input data. The map invocations are distributed across multiple machines by automatically partitioning the input data into a set of splits. The input splits can be processed in parallel by different machines, or be processed in the same machine. Reduce invocations are distributed by partitioning the intermediate key space into pieces using a partitioning function.

From this model, the best known Open Source implementation is Apache Hadoop [6]. Apache Hadoop is a framework used for distributed storage and processing of big data sets. It is written in Java with some native code in C, and it consists mainly of three parts:

1. **HDFS**: Distributed file system that stores data on local disk of commodity hardware.

2. **YARN**: Resource management platform responsible for managing computing resources in clusters, using them for scheduling users applications.

3. **Hadoop MapReduce**: The implementation of the MapReduce programming model.

Hadoop is mainly known because of the MapReduce model and its distributed filesystem (HDFS). However, the name is also used for a family of related projects that fall under the umbrella of its infrastructure for distributed computing and large-scale data processing [26]. This is the so called Hadoop ecosystem. Some examples of these projects are, apart from the already mentioned HDFS, YARN and Hadoop MapReduce: Apache Pig [27], Apache Hive [28], HBase [29], etc. However, all of these projects rely on YARN and the HDFS.

The Hadoop Distributed File System **(HDFS)** is designed to store very large data sets with reliability, and to stream those data sets with a high bandwidth to user applications [23]. HDFS stores the file system metadata separated from the data itself and is rack aware. As in other distributed file systems, like PVFS [30], Lustre [16], or GFS [31, 32], HDFS stores metadata on one or more dedicated servers, called the NameNodes. Application data are stored on other machines, called DataNodes. However, a NameNode process and a DataNode can coexist in the same machine. All servers are fully connected and communicate with each other using TCP-based protocols. The data stored in the DataNodes is divided in blocks (0 to N blocks per DataNode) of a certain size (typically 128 or 64 MB) and with a certain replication factor (the default is 3).

An example of how HDFS works can be seen in Figure 1.2. In this example there are 6 DataNodes and one NameNode. There are 3 files in HDFS (represented in white, red and

blue), each one of them occupying two blocks, with a replication factor of three. Considering this configuration, for example, block 1 of the blue file is stored in the first, third and sixth DataNode, while block two of the same file is stored in DataNodes two, four and five.

Regarding **YARN** (Yet Another Resource Negociator), it is the Hadoop cluster resource management system. It was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms [24].

YARN has a flexible model for making resource requests. A request for a set of containers (processes in the Hadoop terminology) can have different parameters defining the amount of computer resources required for each container (memory and CPU), as well as locality constraints for the containers in that request. It provides its services by using two types of daemons: a Resource Manager (one per cluster) to manage the use of resources across the cluster, and Node Managers running on all the nodes in the cluster (one per node) to launch and monitor containers. A container executes an application specific process with a constrained set of resources. Depending on how YARN is configured, a container may be a Unix process or a Linux cgroup. Figure 1.3 is a modified figure from [6], and illustrates how YARN runs an application.

To execute an application on YARN, a client contacts the Resource Manager and requests it to run an Application Master Process (step 1 in Figure 1.3). The Resource Manager then finds a Node Manager that can launch the Application Master in a container (steps 2a and 2b). Precisely what the Application Master does once it is running depends on the application itself. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the Resource Managers (step 3), and use them to run a distributed computation (steps 4a and 4b).

Finally, locality is critical in ensuring that distributed data processing algorithms use the

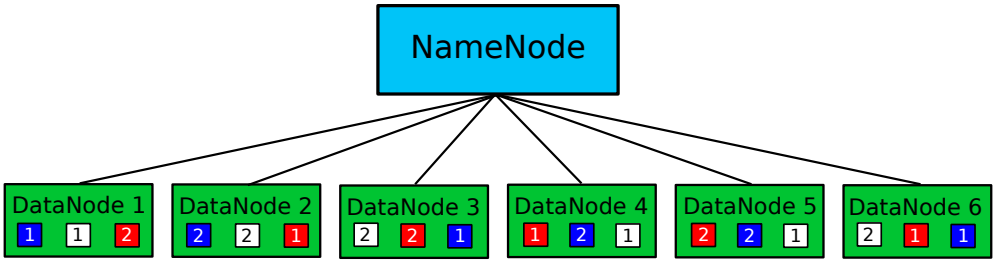


Figure 1.2: Example of how HDFS works.

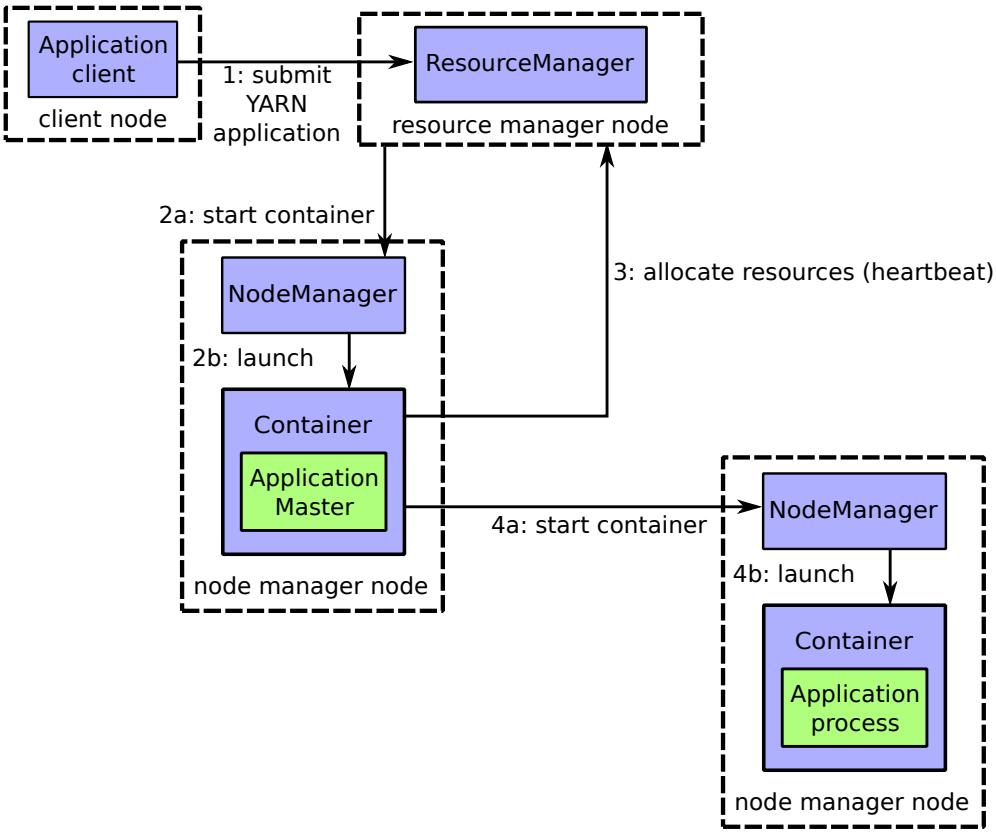


Figure 1.3: Example of how YARN works.

cluster bandwidth efficiently, so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or anywhere on the cluster (off-rack) [33].

1.3. Hadoop Limitations and Apache Spark

One of the main ideas from the HDFS is that the most efficient data processing pattern is a write-once, read-many-times pattern. For this reason, Hadoop shows good performance with embarrassingly parallel applications requiring a single MapReduce execution (assuming intermediate results between map and reduce phases are not huge), and even for applications

requiring a small number of sequential MapReduce executions [34]. Note that Hadoop can also efficiently handle jobs composed by one or more map functions by chaining several mappers followed by a reducer function and, optionally, zero or more map functions, saving the disk I/O cost between map phases. For more complex workflows, solutions as Apache Oozie [35] or Cascading [36], among others, should be used.

The main disadvantage of these workflow managers is the loss of performance when HDFS has to be used to store intermediate data. For example, an iterative algorithm can be expressed as a sequence of multiple MapReduce jobs. Since different MapReduce jobs cannot shared data directly, intermediate results have to be written to disk and read again from HDFS at the beginning of the next iteration, with the consequent reduction in performance. It is worth noting that even each iteration of the algorithm could consist of one or several MapReduce executions. In this case, the degradation in terms of performance is even more noticeable.

Apache Spark is a cluster computing framework designed to overcome the Hadoop limitations in order to support iterative jobs and interactive analytics, originally developed at University of California, Berkeley [13], now managed under the umbrella of the Apache Software Foundation. Spark extends the MapReduce model to efficiently support more types of computations (not only map and reduce), including batch applications, interactive queries, and streaming. One of the main features Spark offers in order to be able to perform this set of computations, is the ability to store the data in main memory between operations.

Another improvement with respect to Hadoop is the programming language. Programs developed to run with Hadoop are written in Java. Hadoop can also run programs written in other languages by using Hadoop Streaming, but some researchers have probed this tool to perform poorly [37]. On the contrary, Spark offers simple APIs in Python, Java, Scala, SQL, and even can be used from R. And not only that. It even includes a rich built-in libraries, for example, for Machine Learning, and also integrates closely with other Big Data tools. In particular, Spark can be run on Hadoop clusters and access any Hadoop data source [38].

The job topology is, however, very similar. A Spark application, at a high level, consists of a driver program that launches various parallel operations on a cluster. The driver program contains the application *main* function and defines **distributed datasets** on the cluster, then applies operations to them. These distributed datasets, commonly named RDDs or Resilient Distributed Dataset (RDD) [39], are one of the Spark main features. A RDD is simply an immutable distributed collection of objects. Each RDD is split into multiple partitions, which

may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user defined classes. RDDs can be created in two ways: by loading an external dataset (for example, from HDFS), or by distributing a collection of objects (e.g., a list or set) in the driver program. Once created, RDDs offer two types of operations: *transformations* and *actions*.

Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. **Actions**, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS). One example of an action is *first()*, which returns the first element in an RDD.

Transformations and actions are different because of the way Spark computes RDDs. Although the user can define new RDDs any time, Spark computes them only in a lazy way, that is, the first time they are used in an action. This approach might seem unusual at first, but makes a lot of sense when working with Big Data. For instance, consider an example where the user defined a text file and then filtered the lines that include "Python". If Spark were to load and store all the lines in the file as soon as possible, it would waste a lot of storage space, given that the user then immediately filter out many lines. Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result. This is due the fact that, in Spark, all the operators in a job are used to construct a DAG (Directed Acyclic Graph). The DAG is optimized by rearranging and combining operators where possible

To run these kind of operations introduced in the previous paragraphs, driver programs typically manage a number of processes in the computing nodes called *executors*. An example of how the Driver Program and the Executors are distributed in Spark can be seen in Figure 1.4.

Finally, Spark's RDDs are by default recomputed each time the programmer runs an action on them. If an RDD is going to be reused in multiple actions, the programmer can ask Spark to persist it using the *persist* method. There are a number of different places where the data can be persisted, for example, memory, disk, memory and disk, etc. This place can be set as an option to the *persist* method. With the *persist* method and no option provided, Spark will store the RDD contents by default in main memory (partitioned across the machines in the computing cluster), and reuse them in future actions. The behaviour of not persisting by default may again seem unusual, but, it makes a lot of sense for big datasets: if the RDD is not going to be reused, there is no reason to waste storage space when Spark could instead stream

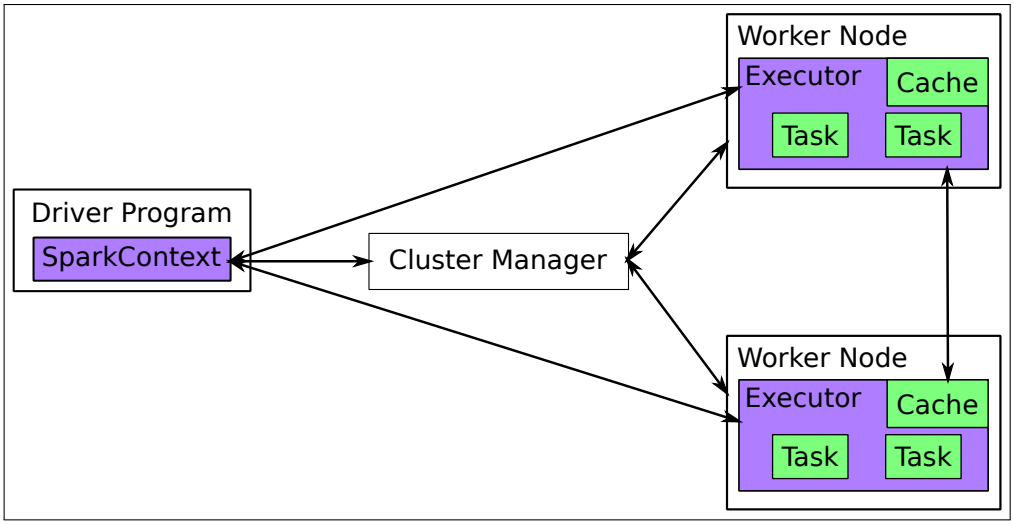


Figure 1.4: Example of how Spark works.

through the data once and just compute the result. These kind of strategies are also useful to implement fault tolerance in Spark. Thanks to the operations DAG, if the RDD fragments stored in one node are lost, they can be rebuild in another node by following the operations graph.

In conclusion, Spark overcomes most of the problems present in Hadoop. However, and despite all of its improvements and parallel philosophy, it is still not clear if Spark fits in the High Performance Computing world.

1.4. Case studies: Natural Language Processing and Genomics

To make progress in the approach between HPC and Big Data technologies, significant applications from the Genomics and Natural Language Processing fields will be designed and integrated into the Big Data frameworks, with the aim of boosting performance and improving scalability.

1.4.1. Natural Language Processing

Natural Language Processing (NLP) is a field of computer science, artificial intelligence and computational linguistics concerned with the interactions between computers and human (natural) languages. In particular, it is concerned with programming computers to fruitfully process large natural language corpora. It is considered as one of the more suitable methodologies to give a structure and organize the textual information accessible through the Internet. Linguistic processing of big quantities of text is a complex task that requires the use of several sub-tasks organized in various inter-related modules that run as a workflow. These modules are usually needed to carry out more complex tasks [40] such as machine translation.

A common task in NLP is the Named Entity Recognition and Classification (NERC). NERC is a research line inside NLP in consolidation phase. The objective of this line is to recognize, identify and classify names inside a text [41, 42, 43]. This process is performed with the help of a predetermined category list (for example, Person, Place, Organization, etc). NERC systems that use linguistic grammar-based techniques or statistical models, such as machine learning, are the state of the art tools. Grammar-based systems typically obtain better precision, but at the cost of lower recall and months of work by experienced computational linguists. Statistical NERC systems typically require a large amount of manually annotated training data. Semi-supervised approaches have been suggested to avoid part of the annotation effort [44, 45]. Many different classifier types have been used to perform machine-learned NERC, with conditional random fields being a typical choice [46].

In this thesis, Linguakit NLP modules have been selected as case study [41, 47, 48]. The NERC module in Linguakit consist of several modules in a pipeline. This NERC system from Linguakit can be observed in Figure 1.5. In these Figure, all the modules involved since the begin of the data process are shown. These modules are:

- **Basic analysis**

    1. Sentence Segmentation: This module divides the input in grammatical sentences.
    2. Tokenizer: Here the sentences are divided into tokens, this is, textual elements separated by white spaces or punctuation marks.
    3. Splitter: The splitter module uses linguistic information to divide those tokens that, despite of being various linguistic units, they are formally represented as an unique element. For example, contractions in spanish, "*del* = *de* + *el*".

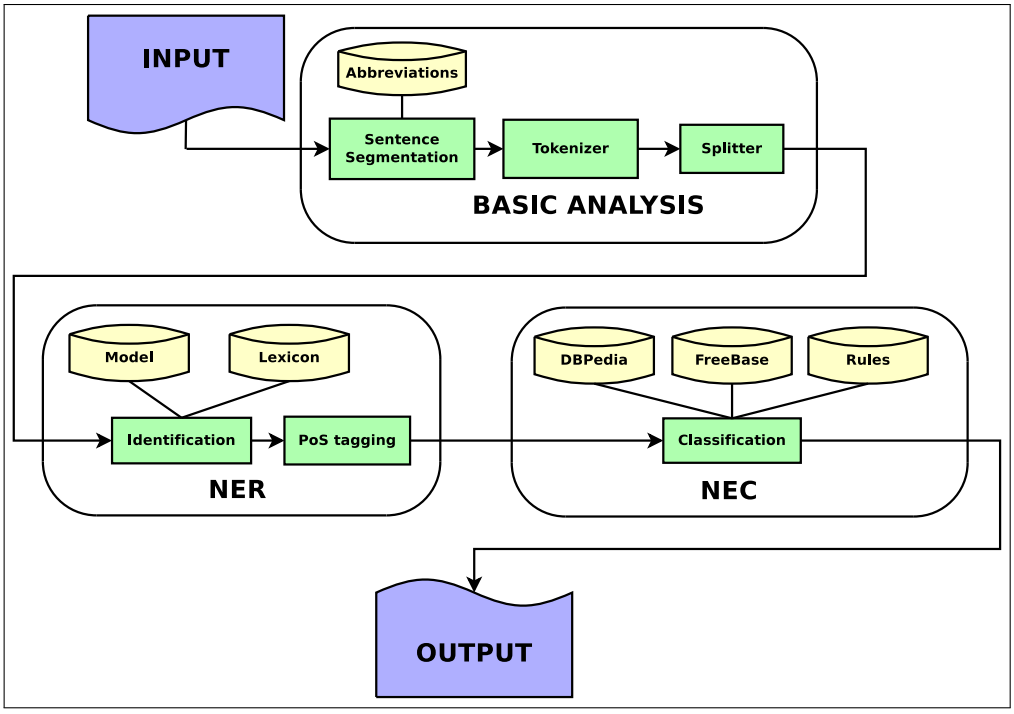


Figure 1.5: NERC system from Linguakit.

- **NER**: Named Entity Recognition

  1. Identification: This task consists of identifying as a single unit (or token) those words or chains of words denoting an entity, e.g. *New York, University of San Diego, Herbert von Karajan*, etc. The module is based on a set of language-independent rules that take into account information on both a large lexicon of forms and the relative position of words within the sentence.
  2. PoS tagging: This module assigns each token of the input text a single PoS tag provided with morphological information e.g. *singular and masculine adjective, past participle verb, plural and feminine noun*, etc. The module consists of a Bayesian classifier whose features are bigrams of tokens that represent the immediate left and right contexts of the target token.

- **NEC**: Named Entity Classification

Le	le	PP3CSD00
dije	decir	VMIS1S0
a	a	SPS00
María	maría	NP00SP0
el	el	DA0MS0
martes	martes	NCMN000
que	que	PR0CN000
me	me	PP1CS000
gusta	gustar	VMIP3S0
el	el	DA0MS0
cubismo	cubismo	NCMS000
.	.	Fp

Table 1.2: Linguakit NERC example.

1. Classification: The last step of the linguistic analysis is the semantic classification of those entities identified in the previous NER module. Named Entity Classification (NEC) is the process of classifying entities by means of classes such as "People", "Organizations", "Locations", or "Miscellaneous". NEC is a crucial task for several natural language applications, for example Question Answering and Information Extraction.

These modules identify and classify the named entities in two phases. First, the identification of token strings that joins to compose the entities (NER), and then the classification by using tags that characterize them in a semantic way. An output example of the Linguakit NERC tool is shown in Table 1.2 for the spanish sentence "*Le dije a María el martes que me gusta el cubismo*". In the example the first column contains the sentence tokens, the second column contains what are called lemmas and the third contains the unique tags which identify the tokens. For example, it identifies that "gusta" is a form of the verb "gustar" and assigns the corresponding tag "VMIP3S0", which starts with a "V" because it is a verb.

Despite all the different approaches that can be found in NERC systems from state of the art tools, all of them lack of the same feature. That is, the capacity to process huge quantities of text in reasonable time. For example, process the whole Wikipedia in Spanish language takes about 19 days (more than 450 hours) [49] when using the considered NERC system. The high execution time proves that one of the biggest problems of these techniques and tools is their high computational cost and scalability problems. For this reason, NLP modules are

non-viable for the analysis of big volumes (GigaBytes or TeraBytes) of documents. In this way, the use of High Performance Computing (HPC) is mandatory, in order to reduce the execution times, improve the system scalability and approaching bigger problems. However, state of the art implementations are written in languages that are not well suited for HPC such as Perl or Python. The reason is that these kind of languages are specially suited to work with text data, so are a good match to NLP. HPC oriented languages such as C or Fortran are more oriented to numerical data.

To overcome this limitation, Hadoop provides the Hadoop Streaming tool, which allows to run a program written in any programming language in Hadoop. However, as was stated before, this tool does not have a good performance [37]. For this reason, **Perldoop** [49] has been developed. The objective in this work was not to develop a powerful tool that allows to automatically translate any existent Perl code to Java, but a simple and easy-to-use tool that takes as input Perl codes written for Hadoop Streaming, following a reduced number of additional programming rules, and produces Hadoop-ready Java codes. In order to do that, Perldoop uses a system based on tags and templates. Templates is contain certain parts of the Java code that has no direct translation from Perl, such as class declarations, some auxiliary functions needed in Java or global variables. Regarding tags, they are used in the Perl code to indicate Perldoop to perform some type of specific translation or decision. The main benefit of using this methodology is the ease of use. Note that programmers have to insert tags in the Perl code and create the templates only once. After that procedure, the Perl code to be translated can be modified at any time. To obtain a new Java version of the code it is only necessary to execute Perldoop again, and it will be automatically generated.

A Hadoop cluster installed at the Galicia Supercomputing Center (CESGA), which consists of 64 nodes has been used in the experimental evaluation. With this infrastructure, an important reduction of the processing time is observed for all the parallel executions with respect to the sequential case both using Hadoop Streaming and Hadoop. For example, the original modules require about 19 days (more than 450 hours) to process the whole Wikipedia, while using Hadoop Streaming this time decreases to bout 16 hours. Despite these important improvements, the execution times using Hadoop Streaming are still high. However, when using the Hadoop-ready codes generated by Perldoop, this time is reduced to less than 2 hours. That is, $8\times$ faster than considering Hadoop Streaming.

More details about the design, implementation and Perldoop performance results can be found in Chapter 2.

1.4.2. Genomics

The first step in a DNA or RNA genomics analysis is always reading a biological sample, taking the genomic information from the sample into a digital format in a computer with the aim of analyzing this data. Emerging next-generation sequencing technologies (NGS) have broken many experimental barriers to genome scale sequencing, facilitating the extraction of huge quantities of sequences, which will further promote the future growth of biological databases. In the last years, the available biological data in a digital format has experimented a big and quick growth. Good examples are the DNA sequence information in the NCBI GenBank [50] database and the protein sequences in the UniProtKB/TrEMBL [51] database.

According to [52], compared to traditional Sanger capillary-based electrophoresis systems, NGS technologies provide ultra-high throughput with a two orders of magnitude lower unit data cost. However, they all share the common intrinsic characteristic of providing short read length, currently 25–75 base pairs (bp), this is, 25-75 characters, which is substantially shorter than the Sanger sequencing reads (500–1000 bp) [53].

These short reads, commonly named sequences, are composed of ASCII characters representing a nucleotide from the sequence, also known as nucleobases or simply, bases. For example, in the DNA case, we can find only four possible bases:

- A - Adenine
- C - Cytosine
- G - Guanine
- T - Thymine

Computer scientists and biomedical researchers face the challenge of transforming these short-reads into biological understanding. Consequently, bioinformatics tools need to be scalable; that is, they need to deal with an ever growing amount of data. Unfortunately, the amount of publicly available sequence data grows faster than the single core processor performance. Thus, modern bioinformatics tools need to take advantage of parallel computing [54].

To give meaning to all this data, according to the GATK [55] best practices from the Broad Institute [56], a wide number of DNA or RNA analysis pipelines perform a pre-processing raw data that comes from high throughput ultra sequencers. To be more precise, they say that "*pre-processing starts from raw sequence data and produces analysis-ready BAM files. This involves* ***alignment*** *to a reference genome*".

1	2	3	4	5	6
AACGT- ACCGTT	-AACGT ACCGTT	A-ACGT ACCGTT	AACGT------ -----ACCGTT	AA-CGT- A-CCGTT	-A-A-C-G-T- A-C-C-G-T-T

Table 1.3: Six different possible alignments for the example sequences.

This pre-processing or alignment is, basically, a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [57]. This alignment of raw data to a reference genome is one of the most time consuming steps in every genomic analysis, and remains as a bottleneck in nowadays workflows Its computing time is very high in comparison with the other steps in the selected workflow. More information about the state of the art algorithms used to perform the alignment phase can be found in [58], [59] or [60].

Short reads alignment

DNA, RNA and protein sequences encode changes in evolution time through mutation. The simplest kinds of mutation are point mutations and insertions/deletions, also known as *indels*. These two types of change are modeled by classical alignment algorithms. As an illustrative example, a set of possible alignments of two DNA sequences, `AACGT` and `ACCGTT`, are shown in Table 1.3 by writing their bases on top of each other, in a way that, either two bases are paired, corresponding to presence or absence of a point mutation, or a base is paired with a gap, corresponding to an insertion or deletion. It is not allowed to pair a gap with a gap, not because two sequences cannot inherit the same deletion (they can), but because there is no enough information to infer such a course of evolution from just two sequences. In Table 1.3, six example alignments that can be generated following these rules are presented [59]. The question now is, which one of the alignments is the "best"? At first sight, alignment 1 may seem the best, as it has four matches, while alignment 2 only has one, alignment 3 has two, and alignments 4 and 6 have no matches. However alignment five also contains four matches, but it contains three gaps, while alignment 1 only has one gap. There are several algorithms to calculate what is called the alignment score. Here, we refer to [58], [59] and [60], where the most common scoring systems are explained in detail.

Putting these two factors together, (the big amount of data generated by sequencers and the high computational cost required to carry out the alignment), help us to understand why the development of parallel tools for the alignment process is so important.

Most of state of the art aligners exploit parallelism on shared memory systems, so they have restrictions regarding the number of cores that can be used. Only some of them take advantage of HPC and Big Data solutions.

- **BWA** [61, 62, 63]: The state of the art tool par excellence regarding sequence alignment is the Burrows-Wheeler Aligner (BWA). It is included in various GATK best practices pipelines and it is widely accepted among the bioinformatics and genomics community. This tool includes thread-level parallelism in shared memory. It is developed in C with three different algorithms available.

- **Bowtie** [64]: Ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. It was developed at the University of Maryland in 2009. As BWA, includes thread level parallelism.

- **SOAP** [65, 66, 67, 68]: Program developed for efficient gapped and ungapped alignment of short oligonucleotides onto reference sequences. It includes GPU support since version 3.

- **Halvade** [69]: Hadoop-based aligner. It includes a variant detection phase which is the next stage after the sequence alignment in some DNA sequencing workflows.

- **SEAL** [70]: Pydoop [71], based aligner. It follows the MapReduce model, and allows users to write their programs in Python, calling BWA methods by means of a wrapper.

- **CUSHAW** [72, 73, 74]: CUDA compatible short read alignment algorithm for multiple GPUs sharing a single host. This aligner is designed based on the Burrows-Wheeler transform (BWT) and written using CUDA C++ parallel programming language.

- **pBWA** [75]: MPI implementation of BWA.

In this work, BWA has been used as the core of two alignment tools developed using Big Data technologies. The first one is BigBWA [76], which takes advantage of Hadoop as Big Data technology to increase the performance of BWA. The main advantages of our tool are the following. First, the alignment process is performed in parallel using a tested and scalable technology, which reduces the execution times dramatically. Second, BigBWA is fault tolerant, exploiting the fault tolerance capabilities of the underlying Big Data technology

on which it is based. And finally, no modifications to BWA are required to use BigBWA. As a consequence, any release of BWA (future or legacy) will be compatible with BigBWA.

Hadoop applications are typically developed in Java, but BWA is implemented in C. To overcome this issue BigBWA uses the Java Native Interface (JNI) [77], which allows the incorporation of native code written in programming languages such as C and C++, as well as code written in Java. Two independent software layers were created in BigBWA. The first one corresponds to the BWA software package, while the other is, strictly speaking, our tool. This design avoids any modification of the BWA source code, which assures the compatibility of BigBWA with any BWA version.

In order to validate our tool several experiments have been carried out on a cluster deployed in Amazon Web Services. Input data correspond to human DNA sequences from the *1000 Genomes* project. Results from these experiments can be observed in Chapter 3, here, let's illustrate the benefits of BigBWA with the following example. The Illumina HiSeqX^{TM} is able to generate 6 billion (6×10^9) reads. It means more than 40 days to perform the alignment phase onto a reference genome with the sequential version of BWA (1 thread). This time can be reduced to 5 days by using the multi-thread BWA version, and less than one day by running BigBWA on a small cluster. In the case of a medium-size cluster, when using BigBWA, this time is reduced to less than 5 hours, it is, 192$\times$ faster than BWA single-thread. At the same time, it is faster than other state of the art tools.

Regarding the second alignment tool developed in this thesis, we have developed the software SparkBWA [78]. SparkBWA follows the same philosophy than BigBWA, but the considered Big Data technology is Spark instead of Hadoop. SparkBWA was designed to meet three requirements. First, SparkBWA should outperform BWA and other BWA-based aligners both in terms of performance and scalability. The second requirement is related to keep the compatibility of SparkBWA with future and legacy versions of BWA. Since BWA is constantly evolving to include new functionalities and algorithms, it is important for SparkBWA to be agnostic regarding the BWA version. This is an important difference with respect to other existent tools based on BWA, which require modifications of the BWA source code. Finally, NGS professionals demand solutions to perform sequence alignments efficiently in such a way that the implementation details are completely hidden to them. For this reason SparkBWA provides a simple and flexible API to handle all the aspects related to the alignment process. In this way, bioinformaticians only need to focus on the scientific problem to deal with.

SparkBWA has been evaluated both in terms of performance, scalability and memory consumption, and a thorough comparison between SparkBWA and several state-of-art BWA-based aligners is also provided. Those tools take advantage of different parallel approaches as Pthreads, MPI, and Hadoop to improve the performance of BWA. Performance results demonstrate the benefits of our proposal. The evaluation shows that SparkBWA is almost twice faster than other state of the art tools. More precisely, it is 1.7× faster than SEAL, 1.4× faster than BigBWA or Halvade, and 1.25× faster than pBWA. SparkBWA reachs a speedup of 86× when using 128 cores regarding the BWA sequential version. Details about the design, implementation and performance results are shown in Chapter 4.

Multiple sequence alignment

The goal of protein sequence comparison is to discover structural or functional similarities among proteins. Biologically similar proteins may not exhibit a strong sequence similarity, but we would still like to recognize resemblance even when the sequences share only weak similarities. If the sequence similarity is weak, pairwise alignment can fail to identify biologically related sequences because weak pairwise similarities may fail statistical tests for significance. However, simultaneous comparison of many sequences often allows one to find similarities that are invisible in pairwise sequence comparison [60]. This is whats is called Multiple Sequence Alignment (MSA). MSA is an extension of the pairwise alignment to incorporate more than two sequences at a time. In many cases, the input set of query sequences are assumed to have an evolutionary relationship by which they share a lineage and are descended from a common ancestor. Multiple sequence alignments are computationally difficult to produce and most formulations of the problem lead to NP-complete combinatorial optimization problems. In this way, MSA is essential in order to predict the structure and function of proteins and RNAs, estimate phylogeny, and other common tasks in sequence analysis.

PASTA [79] is a tool, based on SATé [80], which produces highly accurate alignments, improving the accuracy and scalability of other state-of-art methods. PASTA is based on a workflow composed of several steps. During each phase, an external tool is called to perform different operations such as estimating an initial alignment and tree, computing MSAs on subsets of the original sequence set, or estimating the maximum likehood tree on a previously obtained MSA. Note that computing the MSAs is the most time consuming phase, implying in some cases over 70% of the total execution time.

PASTA is a multithreaded application that only supports shared memory computers. In

this way, PASTA is limited to process small or medium size input datasets, because the memory and time requirements of large datasets exceed the computing power of any shared memory system. In this thesis we introduce PASTASpark [81], an extension to PASTA that allows to execute it on a distributed memory cluster making use of Apache Spark.

We have used the Spark Python API (known as PySpark) to implement PASTASpark. The design of PASTASpark minimizes the changes in the original PASTA code. In fact, the same software can be used to run the unmodified PASTA on a multicore machine or PASTASpark on a cluster.

Regarding the performance results, it is important to remark here that only the most time consuming PASTA phase has been parallelized. The other phases are executed in the Spark driver. With this factor in mind, the speedup obtained in the CESGA Big Data cluster with 64 cores is 10.5$\times$ with respect to the single threaded PASTA. This number seems to be small at first sight, but, actually, it is very close to the upper limit predicted by the Amdahl's law. Our solution is able to process a 200K sequences dataset in less than 24 hours. Considering the original PASTA tool it was not possible to complete this process because of memory restrictions. More details about our approach can be found in Chapter 5.

Some of the state of the art tools to perform MSA using a parallel approach are presented below. To the best of our knowledge, there are no other tools that perform MSA by using Big Data technologies.

- **MAFFT** [82]: MSA program for amino acid or nucleotide sequences. The software is named after the acronym Multiple Alignment using Fast Fourier. Is written in C language and allows multiple threads.

- **MSAProbs** [83]: Tool to perform MSA for protein sequences. It is based on a combination of pair hidden Markov models and partition functions. It is developed in C++ and uses OpenMP to parallelize at thread level with a shared memory approach.

- **MSAProbs-MPI** [84]: MPI implementation of MSAProbs. It implements a distributed memory approach to MSAProbs by using MPI. In this way, it comes with a two-levels parallelism. One in distributed memory by using MPI, and other one in shared memory by using OpenMP.

- **QuickProbs** [85]: GPU based implementation of MSAProbs by using OpenCL.

1.5. Thesis outline

In the following chapters we provide the key articles that represent the main body of work for this thesis. In all these articles, the author of the thesis has been the main contributor. These articles have been either published in JCR journals or in high quality international conferences. The selection of publications has been made to delve into the main points mentioned in this introduction, and to have a more complete representation of the work carried out during this thesis. In Section 1.6, a full compendium of the journal and conference publications related to this thesis is presented.

In Chapter 2 we introduce Perldoop, a new tool developed in order to automatically translate Perl scripts into Hadoop ready Java codes. Our solution is very suitable for Natural Language Processing applications that are written in Perl code. In this way, Perl NLP codes can benefit from the parallel improvements that Hadoop provides. Results prove how using Perldoop the execution times are far better than using Hadoop Streaming with the original Perl codes.

In Chapter 3 we introduce BigBWA, a tool to align raw genomic sequences by using the state of the art software BWA. BigBWA uses Hadoop, HDFS and the MapReduce programming model in order to speed up the alignment process. By using BigBWA important improvements regarding the computation times are observed, while the correctness results compared with the original BWA are almost identical.

An evolution from BigBWA is SparkBWA, which is presented in Chapter 4. SparkBWA presents noticeable improvements in terms of performance and scalability regarding BigBWA and other state of the art tools. Also, it provides a simple and flexible API to handle all the aspects related to the alignment process from the Spark shell and uses Spark native functions to handle the input data.

We introduce PASTASpark in Chapter 5. PASTASpark is an efficient and parallel version of PASTA that uses Spark as Big Data engine. Note that only the alignment step runs into the Spark executors, which is the the most time consuming part in PASTA.

Finally, conclusions future work and some ideas of how the Big Data and the HPC world can converge are presented in Chapter 6.

1.6. List of publications

Next a list of publications derived from the work developed in this thesis is shown:

Articles in peer reviewed journals:

- P. Gamallo, J. C. Pichel, M. Garcia, J. M. Abuín, and T. F. Pena, "Análisis morfosintáctico y clasificación de entidades nombradas en un entorno Big Data," *Procesamiento del Lenguaje Natural*, vol. 53, pp. 17–24, 2014

- J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "BigBWA: Approaching the Burrows–Wheeler Aligner to Big Data Technologies," *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015
  Impact factor (JCR 2015): 5.766. Decil 1 Q1.
  Category: MATHEMATICAL & COMPUTATIONAL BIOLOGY. Rank: **3/56**.

- J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "SparkBWA: speeding up the alignment of high-throughput DNA sequencing data," *PloS one*, vol. 11, no. 5, p. e0155461, 2016
  Impact factor (JCR 2016): 2.806. Q1.
  Category: MULTIDISCIPLINARY SCIENCES. Rank: **15/64**.

- J. M. Abuín, T. F. Pena, and J. C. Pichel, "PASTASpark: multiple sequence alignment meets Big Data," *Bioinformatics*
  Impact factor (JCR 2016): 7.307. Decil 1 Q1.
  Category: MATHEMATICAL & COMPUTATIONAL BIOLOGY. Rank: **2/57**.

Articles published in international conferences:

- J. M. Abuín, J. C. Pichel, T. F. Pena, P. Gamallo, and M. Garcia, "Perldoop: Efficient execution of Perl scripts on Hadoop clusters," in *IEEE International Conference on Big Data*, 2014, pp. 766–771

CHAPTER 2

PERLDOOP: EFFICIENT EXECUTION OF PERL SCRIPTS ON HADOOP CLUSTERS

Following is a reproduction of an article of which the author of this thesis is a main contributor. This is a verbatim reproduction, and the original can be found online with the following information:

> J. M. Abuín, J. C. Pichel, T. F. Pena, P. Gamallo, and M. Garcia, "Perldoop: Efficient execution of Perl scripts on Hadoop clusters," in *IEEE International Conference on Big Data*, 2014, pp. 766–771

2.1. Abstract

Hadoop is one of the most important implementations of the MapReduce programming model. It is written in Java and most of the programs that run on Hadoop are also written in this language. Hadoop also provides an utility to execute applications written in other languages, known as Hadoop Streaming. However, the ease of use provided by Hadoop Streaming comes at the expense of a noticeable degradation in the performance.

In this work, we introduce Perldoop, a new tool that automatically translates Hadoop-ready Perl scripts into its Java counterparts, which can be directly executed on Hadoop while improving their performance significantly. We have tested our tool using several Natural Language Processing (NLP) modules, which consist of hundreds of regular expressions, but

Perldoop could be used with any Perl code ready to be executed with Hadoop Streaming.

Performance results show that Java codes generated using Perldoop execute up to 12x faster than the original Perl modules using Hadoop Streaming. In this way, the new NLP modules are able to process the whole Wikipedia in less than 2 hours using a Hadoop cluster with 64 nodes.

2.2. Introduction

In the modern digital society, it is estimated that each day are created around 2.5 quintillion bytes of data (2.5 Exabytes), in such a way that 90% of the data all over the world were created just only in the last two years [1]. These data come from all type of sources: sensors used to obtain information on the climate, publications in social networks, blogs, digital images and video, etc. For instance, Twitter generates about 8 Terabytes of data per day, while Facebook captures about 500 Terabytes. This is what is known as Big Data. One of the main characteristics of this amount of information is the fact that, in many cases, is not structured.

The MapReduce framework has become the de-facto standard for parallel processing of Big Data and has gained a wide adoption in both industry and research fields. One of the most successful open-source implementations based on Google's MapReduce [5] programming model is Hadoop [6], which is implemented using Java. In this model, the input and output of a MapReduce computation is a list of *(key, value)* pairs. Users only need to implement *Map* and *Reduce* functions. Each *map* produce zero or more intermediate *(key, value)* pairs by consuming one *(key, value)* pair. After this, the runtime groups automatically these intermediate *(key, value)* pairs into buckets representing *reduce* tasks. *Reduce* functions take an intermediate key and a list of values as input and produce zero or more output results.

Even though code developing in Hadoop is largely simplified with its characteristics as the automatic input splitting, task scheduling or fault tolerance mechanism, to write a Java MapReduce program is not straightforward. Besides, in some research areas, Java is not normally employed, and programmers are more familiar with other high level programming languages like Perl or Python. For example, Natural Language Processing (NLP) and Bioinformatics researchers are used to write code in Perl due to its unique ability to process text using regular expressions. These researchers have found in Hadoop Streaming the way to easily analyze big volumes (Gigabytes or even Terabytes) of textual information. However, important degradations in the performance were detected using Hadoop Streaming with re-

spect to Hadoop Java codes [37]. Only for computational intensive jobs whose input/output size is small, the performance of Hadoop Streaming is sometimes better because of using a more efficient programming language.

For the reasons detailed above, in this paper we introduce Perldoop, a new tool that automatically translates Perl scripts prepared to be executed using Hadoop Streaming into Hadoop-ready Java codes. Our tool has been tested using several NLP modules as input, which consist of hundreds of regular expressions. In particular, three linguistic modules were considered: Name Entity Recognition (NER), PoS-Tagging and Named Entity Classification (NEC).

Performance tests were carried out on a 64 nodes Hadoop cluster. The results show that the automatically generated Java codes execute up to 12× faster than the original Perl modules using Hadoop Streaming. In this way, the new NLP modules are able to process the whole Wikipedia in less than 2 hours.

2.3. The Perldoop tool

As it was stated in the introduction, our objective is to translate Hadoop Streaming codes written in Perl to its Java equivalents, in order to take advantage of the higher performance of Java codes in Hadoop [37]. The general case of automatically translating an arbitrary Perl code into its Java equivalent is a very hard problem, due to the characteristics of both languages. One of the main difficulties to create a general source-to-source translator is that Perl has a Turing-complete grammar in such a way that parsing can be affected by run-time code executed during the compile phase. Therefore, Perl cannot be parsed by a straight Lex/Yacc lexer/parser combination. Instead, the interpreter implements its own lexer, which coordinates with a modified GNU Bison parser to resolve ambiguities in the language [87, 88].

Some efforts have been done to integrate (not to translate) Perl into Java code. This is the case of the Java-Perl Library (JPL) [89], which allows to invoke Perl methods inside a Java program. We discard this solution because the performance obtained by JPL codes is equivalent to the one obtained by using directly Perl codes and Hadoop Streaming.

On the other hand, we cannot forget that Perl and Java are two very different languages. There are a lot of differences between them, but the following are the most relevant to our case:

- **Variable declaration:** Programmers do not have to declare and establish the variable

type in Perl, while that is mandatory in Java.

- **Array size and accesses:** If the programmers want to access a non existent array position in Perl, the array is expanded if it is a write operation, and positions in the middle are set to *undef*, or, in the case of reading, it returns *undef* [90]. However, Java produces an execution error.

- **Boolean values:** Perl does not have boolean values such as "True" and "False", which can be assigned to variables. Instead, it can handle another variables as booleans. For example, the strings "" and "0" are considered as "False", and also the integer and real values 0 and 0.0 [90]. Java uses boolean variables, and it cannot process integers, real or strings as booleans like Perl.

Due to all the aforementioned difficulties, our objective in this work was not to develop a powerful tool that allows to automatically translate any existent Perl code to Java, but a simple and easy-to-use tool that takes as input Perl codes written for Hadoop Streaming, following a reduced number of additional programming rules, and produces Hadoop-ready Java codes. We have called this tool Perldoop, and it was developed using Python.

2.3.1. How it works

Perldoop uses file templates and tags, as it is shown in Figure 2.1. The main goal of the templates is to contain certain parts of the Java code that has no direct translation from Perl, such as class declarations, some auxiliary functions needed in Java or global variables. In the near future, and as Perldoop evolves, we expect that templates will not be necessary.

The programmer must indicate the position to insert the translated code into the file template using the next tags:

`//<java><start>` and `//<java><end>`

In the same way, the Perl code to be translated needs to be surrounded by the following tags:

`#<perl><start>` and `#<perl><end>`

The translation process can be summarized in these steps:

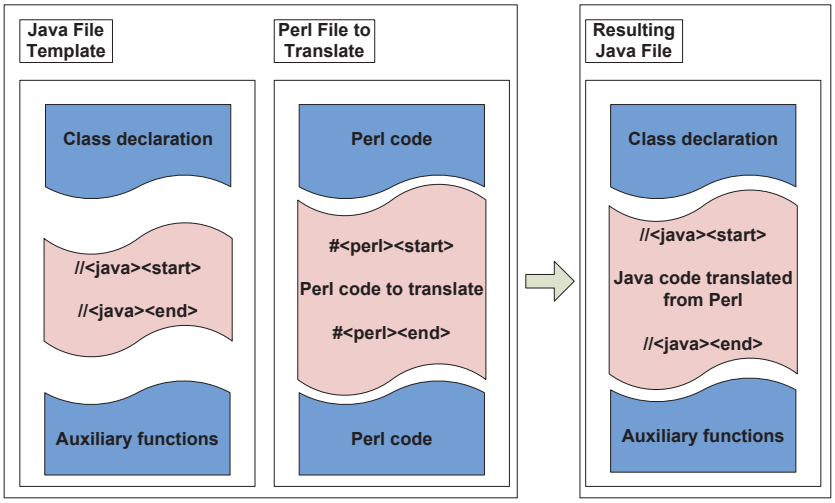


Figure 2.1: Use of templates and tags with Perldoop.

1. The programmer tags the Perl file and creates the Java template, including the class declaration and constructor.

2. The Perldoop tool is executed to generate the new Java code.

The main benefit of using this methodology is the simplicity of use. Note that programmers have to insert the labels in the Perl code and create the templates only once. After that procedure, the Perl code to be translated can be modified at any time. To obtain a new Java version of the code it is only necessary to execute Perldoop again, and it will be automatically generated.

Regarding the limitations of Perldoop, the main one is that programmers have to tag the Perl code and make the corresponding Java template. In addition, there are a few tips or rules that the programmer should follow in order to guarantee the correct translation when using Perldoop.

2.3.2. Programming rules

Next we detail the programming rules that the Perl codes should follow to assure the correctness of the Java codes automatically generated by Perldoop:

1. Ordered conditional blocks. It means that the conditional expression should appear before the sentences to be executed if the condition is fulfilled. Use:

   ```
   if(condition){
   sentences;
   }
   ```

   instead of: `sentences if(condition);`

2. Perform string concatenations with the "." operator. For example:

   ```
   $variable = $var1." ".$var2;
   ```

3. Restrict the access to array positions not previously allocated.

4. Use a different name for each variable. Perl allows:

   ```
   my $feat; # variable
   my @feat; # array
   ```

   while in Java the programmer has to declare two different variables.

5. In Perl, the programmer can do the following using an integer variable: `if($variable)`

   Java only allows this expression if the variable is a boolean. Therefore, the expression should be:

   ```
   if($variable!=0)
   ```

   A similar situation arises when using hashes in Perl.

6. Declare and initialize the variables with the corresponding label, which includes the data type and class (variable, array, etc.). For example:

   ```
   #<var><string>
   #<array><integer>
   #<arraylist><string>
   ```

```perl
#!/usr/bin/perl -w

#<perl><start>

my $line;               #<var><string>
my @words;              #<array><string>
my $key;                #<var><string>
my $valueNum = "1";     #<var><string>
my $val;                #<var><string>

while ($line = <STDIN>) {       #<map>
    chomp ($line);
    @words = split ("␣",$line);
    foreach my $w (@words) {    #<var><string>
        $key = $w."\t";
        $val = $valueNum."\n";
        print $key.$val;
    }
}

#<perl><end>
```

```java
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public static class WordCountMap extends Mapper<
      Object, Text, Text, Text>{
    @Override
    public void map(Object incomingKey, Text
          value,
    Context context) throws IOException,
          InterruptedException {
        try{

        //<java><start>
        String line;
        String[] words;
        String key;
        String valueNum  = "1";
        String val;
        line  = value.toString();
        line = line.trim();
        words  = line.split("␣");
        for (String w : words) {
            key  = w+;
            val  = valueNum;
            context.write(new Text(key),new Text(
                  val));
        }
        //<java><end>

        }
        catch(Exception e){
            System.out.println(e.toString());
        }
    }
}
```

Figure 2.2: WordCount mapper example using Perl (left) and its equivalent Java code generated using Perldoop (right).

In addition, we must take into account that Boolean values are not available in Perl. The next label is used to identify a boolean variable:

```
#<var><boolean>
```

Additionally, programmers should also include a label to indicate if the Perl code corresponds to a mapper (`<map>`) or a reducer (`<reduce>`).

As we have commented previously, Perl is well-known for its unrivaled ability to process text using very powerful features such as regular expressions. The native Java support for regular expressions is not as good as the provided by Perl. A list of the main differences can be found in [91]. For this reason, in order to improve the handle of regular expressions in Java, Perldoop takes advantage of the *jregex* library [92]. This library uses Perl 5.6 regex syntax, including lookahead/lookbehind assertions and it holds a BSD license.

```perl
#!/usr/bin/perl -w

#<perl><start>

my $count = 0;          #<var><integer>
my $value;              #<var><integer>
my $newkey;             #<var><string>
my $oldkey;             #<var><string><null>
my $line;               #<var><string>
my @keyValue;           #<var><string>

while ($line = <STDIN>) {       #<reduce>
  chomp ($line);
  $keyValue = split ("\t",$line);

  $newkey = $keyValue[0];
  $value  = $keyValue[1];

  if (!(defined($oldkey))) {
    $oldkey = $newkey;
    $count  = $value;
  }
  else {
    if ($oldkey eq $newkey) {
      $count = $count + $value;
    }
    else {
      my $returnKey = $oldkey."\t";#<var><string>
      my $returnValue = $count."\n";#<var><string>
      print $returnKey.$returnValue;
      $oldkey = $newkey;
      $count  = $value;
    }
  }
}
my $returnKey = $oldkey."\t";   #<var><string>
my $returnValue = $count."\n";  #<var><string>
print $returnKey.$returnValue;

#<perl><end>
```

```java
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public static class WordCountReducer extends
      Reducer<Text, Text, Text, Text> {
  int count  = 0;
  @Override
  public void reduce(Text key, Iterable<Text>
       values,
  Context context) throws IOException,
       InterruptedException {

    //<java><start>
    int value;
    String newkey;
    String oldkey = null;
    String line;
    for (Text val : values) {
      newkey    = key.toString();
      value     = Integer.parseInt(val.toString()
           );
      if (!(oldkey!= null)) {
        oldkey  = newkey;
        count   = value;
      }
      else{
        if (oldkey.equals(newkey) ) {
          count  = count + value;
        }
        else{
          String returnKey    = oldkey;
          String returnValue  = String.valueOf(
               count);
          context.write(new Text(returnKey),
          new Text(returnValue));
          oldkey  = newkey;
          count   = value;
        }
      }
    }
    String returnKey    = oldkey;
    String returnValue  = String.valueOf(count);
    context.write(new Text(returnKey),new Text(
         returnValue));
    //<java><end>

  }
}
```

Figure 2.3: WordCount reducer example using Perl (left) and its equivalent Java code generated using Perldoop (right).

2.3.3. Example: WordCount in Perl

Next we present, for illustrating purposes, an example of the use of Perldoop. The goal is to translate a simple Perl script into a Hadoop-ready Java class. In particular, we have selected the Perl version of the WordCount, which counts the number of occurrences of each word in a text document. This code is a typical example used to test a Hadoop infrastructure.

Figures 2.2 and 2.3 show, to the left, the Perl code of the WordCount mapper and reducer,

respectively. Note that these codes have been tagged following the programming tips detailed above. Next, the programmer has to create the Java templates with the class declaration where the translated codes will be inserted. Note that templates correspond to the Java code not included between <java><start> and <java><end>. After this step, Perldoop is executed. The resulting Hadoop-ready Java codes are displayed to yhe right of Figures 2.2 and 2.3. We must highlight that Perldoop can be applied to any Perl code ready to be executed with Hadoop Streaming.

2.4. Case studies: NLP scripts

Natural Language Processing (NLP) is considered as one of the methodologies more suited to structure and organize the textual information accessible through Internet. Linguistic processing of large amount of text is a complex task that requires the use of several subtasks organized in interconnected modules. Most of the existent NLP modules are programmed using Perl due to its unique ability to process text using regular expressions. One of the main problems found by the researchers in the NLP area is the high computational cost of their tools, which makes them impractical for the analysis of big volumes (Gigabytes or even Terabytes) of documents. As a consequence, the use of parallelism and Big Data technologies is mandatory in order to overcome these limitations.

Perldoop has been tested using several NLP modules. In particular, we have used a set of linguistic modules [93, 41]. These modules are written in Perl, and perform accurate linguistic annotation on large amounts of text corpora. The whole architecture consists of a pipeline in which these modules are chained, in such a way that the output of each module feeds directly the input of the next one. The text is linguistically annotated at increasingly complex level of analysis, i.e. sentence segmentation, tokenization, word splitting, Named Entity Recognition (NER), Part-of-Speech (PoS) tagging, and Named Entity Classification (NEC). The analysis chain has more than 150 regular expressions. In this section, we will briefly describe the last three processes in the analysis chain: NER, PoS-tagging, and NEC. These modules have been integrated into a Hadoop infrastructure using Perldoop. Note that these modules are suited to be used in more complex and higher level linguistic applications such as machine translation, information retrieval, question answering, or even new intelligent systems for technological surveillance and monitoring.

- *Named Entity Recognition (NER)*: This task consists of identifying as a single unit (or token) those words or chains of words denoting an entity, e.g. *New York*, *University*

of San Diego, *Herbert von Karajan*, etc. The module is based on a set of language-independent rules that take into account information on both a large lexicon of forms and the relative position of words within the sentence.

- *PoS-Tagging*: This module assigns each token of the input text a single PoS tag provided with morphological information e.g. *singular and masculine adjective*, *past participle verb*, *plural and feminine noun*, etc. The module consists of a Bayesian classifier whose features are bigrams of tokens which represent the immediate left and right contexts of the target token [93][94]. The module makes use of the same tagset and lexicon as FreeLing, a well-known suite of multilingual linguistic tools [95].

- *Named Entity Classification (NEC)*: The last step of the linguistic analysis is the semantic classification of those entities identified in the previous NER step. Named Entity Classification (NEC) is the process of classifying entities by means of classes such as "People", "Organizations", "Locations", or "Miscellaneous". NEC is a crucial task for several natural language applications, namely Question Answering and Information Extraction. The NEC module relies on a distant-supervised strategy and consists of two tasks. First, large resources (e.g. gazeetters of persons, locations, and organizations) are automatically generated with the aid of encyclopaedic data stored in databases such as FreeBase [96] and DBpedia [97]. Second, a set of disambiguation rules are applied on previously identified entities, in order to solve both ambiguous and unknown entities. This module was described in [41] and reached state-of-the-art precision on different test corpora.

2.5. Performance evaluation

The experiments shown in this section were performed on a Hadoop cluster installed at the Galicia Supercomputing Center (CESGA), which consists of 64 nodes. Each node has an Intel Xeon E5520 processor and 1 GB of RAM memory. The Hadoop version is the 1.1.2, while the Java and Perl versions are 1.7.0 and 5.10.1 respectively. The performance results for the NLP Perl modules considered in the work (NER, Tagger and NEC) were obtained using the Wikipedia in plain text (file size of 2.1 GB) as input. The block size was 128 MB.

As a first approach, we have integrated the sequential NLP Perl modules into a Hadoop infrastructure using the Hadoop Streaming tool. Afterwards, the same Perl codes were auto-

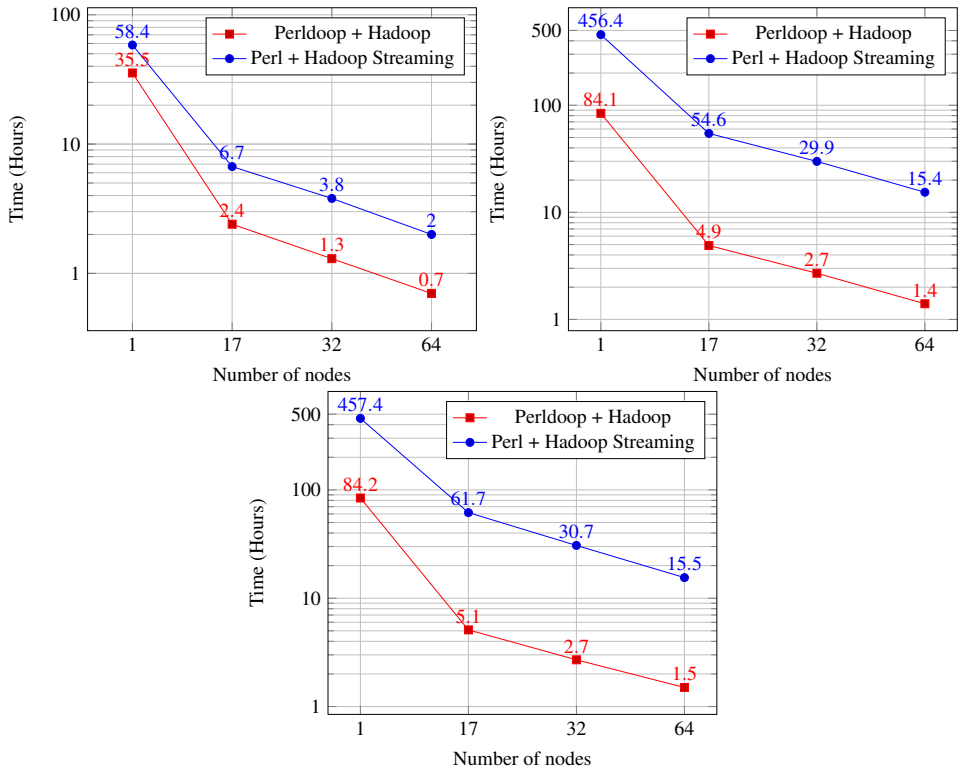


Figure 2.4: Execution time of the NER (top left), Tagger (top right) and NEC (bottom) modules on a Hadoop cluster (log scale).

matically converted into Hadoop-ready Java codes using the Perldoop tool. A performance comparison of both approaches is shown next. Note that in this particular case only mappers were generated because reducers are not necessary.

Figures 2.4 shows the execution times of the NER, Tagger and NEC modules on the cluster using both, Hadoop Streaming and the new automatically generated Java codes with Hadoop (Perldoop + Hadoop in the figures). Different number of nodes were considered. Note that 17 nodes were used instead of 16 because, for the latter, the split size was bigger than the block size. An important reduction of the processing time is observed for all the parallel executions with respect to the sequential case, both using Hadoop Streaming and Hadoop. For example, the original Tagger and NEC modules require about 19 days (more than 450 hours) to process the whole Wikipedia, while using Hadoop Streaming this time reduces to less than 16 hours

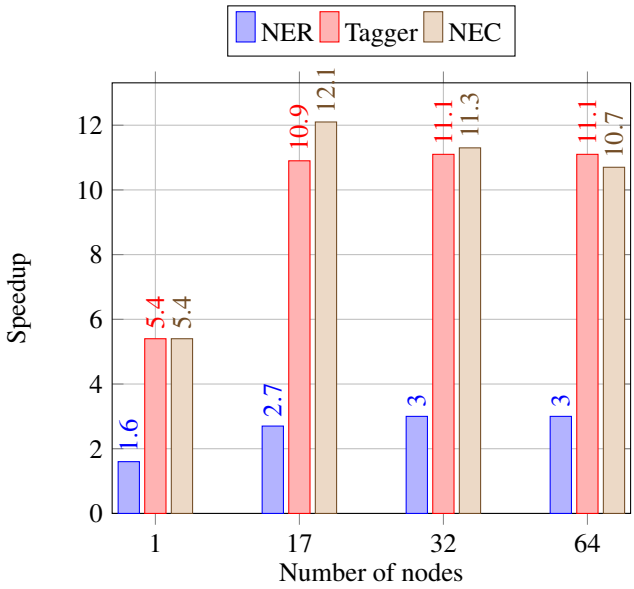


Figure 2.5: Performance improvement of the Java modules generated by Perldoop using Hadoop with respect to the use of Perl and Hadoop Streaming.

using 64 nodes. Despite these important improvements, the execution times using Hadoop Streaming are still too high.

Java codes generated by Perldoop using one node behave better than their Perl counterparts, but the processing times are also very high. Considering the parallel executions, the performance of the new Java modules clearly outperforms the Perl ones, reducing the processing times to less than 2 hours for all the NLP modules when using 64 nodes.

Figure 2.5 shows the speedups obtained by the Perldoop generated Java modules using Hadoop with respect to the original Perl codes using Hadoop Streaming. The performance gains range from 1.6× to 12.1×. For parallel executions, we must highlight that the Tagger and NEC modules process the Wikipedia, at least, 10× faster than using Hadoop Streaming. The worst behavior is observed for the NER module, which is the NLP module with low computational cost. Despite of that, speedups up to 3× are reached. These results confirm the important performance differences between using Java codes with Hadoop, and the Perl modules with Hadoop Streaming.

But, in addition to decrease the processing times, the scalability of the new NLP modules

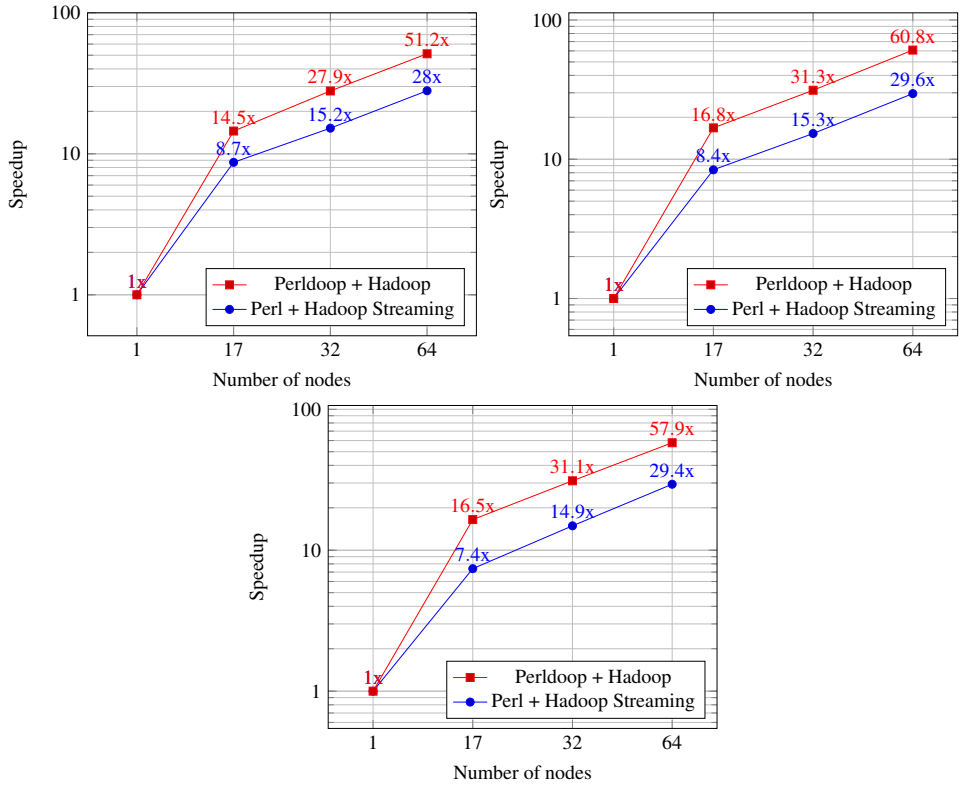


Figure 2.6: Speedup with respect to the sequential version in Java and Perl for the NER (top left), Tagger (top right) and NEC (bottom) modules (log scale).

also improves when using Hadoop. Figure 2.6 shows the speedup of the Java and Perl versions of the modules when using Hadoop and Hadoop Streaming. While the speedups observed with Hadoop Streaming are far from the ideal case, the new Java modules generated by Perldoop are closer to it. For example, considering 64 nodes, the speedups of the NER, Tagger and NEC modules are 51.2×, 60.8× and 57.9× respectively.

2.6. Related work

With respect to the Hadoop performance, several studies have compared Hadoop Streaming with pure Hadoop Java codes and probed that Hadoop Streaming degrades the performance a lot for data intensive jobs [37]. Other authors proposed improvements to the Hadoop

Streaming code [98] or developed their own MapReduce framework on Hadoop [99]. Alternatively, in [71] a Python based programming model for MapReduce similar to the Java one is presented, which provides a Python API for both the MapReduce and the distributed file system using Hadoop Pipes. In any case, despite the improvement gained over Hadoop Streaming, Java codes still have a better performance in Hadoop.

In the last few years, some work has been carried out to use Big Data technologies (mainly the MapReduce programming models) to deal with some aspect of NLP. In statistical translation, MapReduce has been used in [100] and [101]. In [102], the author uses Hadoop to build word co-occurrence matrices from large corpora, whilst Pantel et al. [103] use the MapReduce framework for computing the pairwise semantic similarity between words and in [104] the authors use it for paraphrase acquisition. Most of these works developed ad-hoc solutions adapted to the MapReduce paradigm, using Java in order to get the best performance.

Other authors proposed to adapt existing codes, written in scripting languages like Python, to the MapReduce framework. So, in [105], Hadoop Pipes and SWIG [106] have been used to integrate NLTK (*Natural Language Toolkit*) [107], which is written in Python, into Hadoop. Hadoop Pipes provides slightly better performance than Streaming, but it is worse than Java. On the other hand, Attardi et al. [108], present a suite of tools for text analytics based on the software architecture paradigm of data pipelines, using a modified version of Hadoop Streaming that allows them to have an ordered output. Unlike those works, the solution presented in this paper uses previously developed Perl codes, which effortlessly are translated into Java code ready to be executed in Hadoop. So, it combines the expressiveness and power of Perl regular expressions with the good performance of Java codes running in Hadoop.

2.7. Conclusions

Hadoop is the most important implementation of the MapReduce programming model. It provides an utility to execute applications written in languages different from Java, known as Hadoop Streaming. However, the ease of use provided by Hadoop Streaming comes at the expense of noticeable degradations in the performance.

In this work, we introduce Perldoop, a new tool that automatically translates Hadoop Streaming scripts written in Perl into Hadoop-ready Java codes. Perldoop is a simple and easy-to-use tool that takes as input Perl codes written following a reduced number of programming rules, and produce Hadoop-ready Java codes. To the best of our knowledge, this is the first tool to deal with this problem.

Perl is well-known for its unrivaled ability to process text using very powerful features like regular expressions. For this reason, a lot of Natural Language Processing (NLP) applications have been developed using this language. We have tested our tool using several NLP modules that perform accurate linguistic annotation on large amounts of text corpora. In particular, the linguistic modules considered in this work carry out the following tasks: Named Entity Recognition (NER), Part-of-Speech (PoS) tagging, and Named Entity Classification (NEC). These modules were automatically translated into Hadoop-ready Java codes using Perldoop.

A performance comparison using the original Perl scripts with Hadoop Streaming, and the new Java codes with Hadoop was performed on a cluster. An important decrease in the processing times was observed with respect to the sequential case, both using Hadoop Streaming and Hadoop. However, the performance of the new Java modules clearly outperforms the Perl ones, reaching speedups up to $12\times$. We must highlight that the new Java modules reduce the time required to process the whole Spanish Wikipedia to less than 2 hours when using 64 nodes, demonstrating the benefits of using Perldoop.

Chapter 3

BigBWA: Approaching the Burrows-Wheeler Aligner to Big Data Technologies

Following is a reproduction of an article of which the author of this thesis is a main contributor. This is a verbatim reproduction, and the original can be found online under the following DOI: *10.1093/bioinformatics/btv506*, or with this information:

J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "BigBWA: Approaching the Burrows–Wheeler Aligner to Big Data Technologies," *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015

3.1. Abstract

BigBWA is a new tool that uses the Big Data technology Hadoop to boost the performance of the Burrows-Wheeler Aligner (BWA). Important reductions in the execution times were observed when using this tool. In addition, BigBWA is fault tolerant and it does not require any modification of the original BWA source code.

All the datasets were extracted from the 1000 Genomes Project [109].

Tag	Name	Number of reads	Read length	Size
D1	NA12750/ERR000589	12×10^6	51 bp	3.9 GB
D2	HG00096/SRR062634	24.1×10^6	100 bp	13.4 GB
D3	150140/SRR642648	98.8×10^6	100 bp	54.7 GB

Table 3.1: Main characteristics of the input datasets.

3.2. Introduction

Burrows-Wheeler Aligner (BWA) is a very popular software for mapping sequence reads to a large reference genome. It consists of three algorithms: BWA-backtrack [61], BWA-SW [62] and BWA-MEM [63]. The first algorithm is designed for short Illumina sequence reads up to 100bp, while the others are focused on longer reads. BWA-MEM, which is the latest, is preferred over BWA-SW for 70bp or longer reads as it is faster and more accurate. In addition, BWA-MEM has shown better performance than other several state-of-art read aligners for mapping 100bp or longer reads.

Sequence alignment is a very time-consuming process. This problem becomes even more noticeable as millions and billions of reads need to be aligned. For instance, new sequencing technologies, such as Illumina HiSeqX® Ten, generate up to 6 billion reads per run, requiring more than 4 days to be processed by BWA on a single 16-core machine. Therefore, NGS professionals demand scalable solutions to boost the performance of the aligners in order to obtain the results in reasonable time.

In this paper we introduce BigBWA, a new tool that takes advantage of Hadoop as Big Data technology to increase the performance of BWA. The main advantages of our tool are the following. First, the alignment process is performed in parallel using a tested and scalable technology, which reduces the execution times dramatically. Second, BigBWA is fault tolerant, exploiting the fault tolerance capabilities of the underlying Big Data technology on which it is based. And finally, no modifications to BWA are required to use BigBWA. As a consequence, any release of BWA (future or legacy) will be compatible with BigBWA.

3.3. Approach

BigBWA uses Hadoop as Big Data technology. Hadoop is the most successful open-source implementation of the MapReduce programming model introduced by Google [5]. Hadoop applications are typically developed in Java, but BWA is implemented in C. To overcome this issue BigBWA takes advantage of the Java Native Interface (JNI) [77], which allows

the incorporation of native code written in programming languages such as C and C++, as well as code written in Java. Two independent software layers were created in BigBWA. The first one corresponds to the BWA software package, while the other is, strictly speaking, our tool. This design avoids any modification of the BWA source code, which assures the compatibility of BigBWA with any BWA version.

The complete BigBWA workflow consists of four steps: convert the *fastq* input files to a Hadoop compatible format, copy the input data to the Hadoop cluster (HDFS), perform the alignment, and copy the output back from HDFS to the local filesystem. For more derails, refer to the Supplementary Material.

Regarding the alignment process, BigBWA divides the computation into Map and Reduce phases. In the Map phase, BigBWA splits the reads into subsets, mapping each subset to a mapper process. Each mapper is responsible for applying the considered BWA algorithm using as input the reads assigned by BigBWA. Mappers are processed concurrently, speeding up the alignment process. In case any of the mappers fails, BigBWA would automatically launch another identical mapper process to replace the faulty one. At the end, BigBWA generates one output file per mapper. In the reducer phase those files are merged into one unique solution. Note that users could choose to skip the reduction phase.

Highlighted the best tool for a particular number of cores. For fair comparison with the other tools, BigBWA obtains these results using BWA version 0.5.10. Tool versions: pBWA 0.5.9 and SEAL 0.4.0.

Dataset	**Tool**	**Execution time (minutes)**						**Speedup**				
		Number of cores						Number of cores				
		1	4	8	16	32	64	4	8	16	32	64
D1	SEAL	148.5	55.7 ± 1.6	28.3 ± 1.0	22.2 ± 0.6	11.1 ± 0.1	5.7 ± 0.0	2.7	5.2	6.7	13.4	26.0
	pBWA		**42.0 ± 0.7**	25.3 ± 1.1	17.7 ± 0.5	9.2 ± 0.1	5.1 ± 0.1	**3.5**	5.9	8.4	16.1	29.1
	BigBWA		42.4 ± 0.9	**23.8 ± 0.7**	**15.4 ± 0.4**	**8.5 ± 0.2**	**4.5 ± 0.1**	3.5	**6.2**	**9.6**	**17.3**	**33.0**
D2	SEAL	556.9	186.5 ± 1.7	92.6 ± 0.8	68.1 ± 1.9	35.4 ± 0.7	18.5 ± 0.3	2.9	6.0	8.2	15.7	30.1
	pBWA		155.0 ± 0.4	94.5 ± 1.6	61.2 ± 1.5	32.7 ± 0.4	17.1 ± 0.3	3.6	5.9	9.0	17.0	32.6
	BigBWA		**152.0 ± 0.3**	**82.3 ± 1.6**	**57.2 ± 0.8**	**30.3 ± 0.5**	**15.3 ± 0.1**	**3.7**	**6.8**	**9.7**	**18.3**	**36.4**

Table 3.2: Comparison of the performance for the BWA-backtrack algorithm.

Highlighted the best tool for a particular number of cores. These results were obtained using BWA version 0.7.12.

Dataset	**Tool**	**Execution time (minutes)**						**Speedup**				
		Number of cores						Number of cores				
		1	4	8	16	32	64	4	8	16	32	64
D1	BWA-Threads	106.6	**27.6 ± 0.1**	**14.3 ± 0.1**	10.9 ± 0.0	–	–	**3.9**	**7.4**	9.8	–	–
	BigBWA (hybrid)		29.6 ± 0.2	15.1 ± 0.3	11.8 ± 0.1	6.8 ± 0.3	3.6 ± 0.1	3.6	7.0	9.0	15.7	29.6
	BigBWA		29.1 ± 0.3	15.7 ± 0.1	**7.9 ± 0.1**	**4.5 ± 0.1**	**3.0 ± 0.1**	3.7	6.8	**13.4**	**23.5**	**35.5**
D2	BWA-Threads	258.0	**66.0 ± 0.1**	**33.7 ± 0.1**	24.9 ± 0.0	–	–	**3.9**	**7.6**	10.4	–	–
	BigBWA (hybrid)		69.6 ± 1.3	36.4 ± 0.6	24.5 ± 0.5	15.3 ± 0.1	8.8 ± 0.1	3.7	7.1	10.5	16.8	29.3
	BigBWA		69.1 ± 1.4	37.5 ± 0.4	**20.7 ± 0.5**	**10.9 ± 0.3**	**7.2 ± 0.3**	3.7	6.9	**12.5**	**23.6**	**35.8**
D3	BWA-Threads	3208.6	**816.8 ± 2.5**	**408.1 ± 1.7**	333.3 ± 0.3	–	–	**3.9**	**7.9**	9.6	–	–
	BigBWA (hybrid)		828.8 ± 9.5	431.1 ± 9.0	**221.9 ± 4.0**	183.3 ± 2.2	107.2 ± 0.8	3.9	7.4	**14.5**	17.5	29.9
	BigBWA		848.8 ± 13.6	444.9 ± 8.2	229.2 ± 5.1	**120.1 ± 1.4**	**87.8 ± 0.2**	3.8	7.2	14.0	**26.7**	**36.6**

Table 3.3: Comparison of the performance for the BWA-MEM algorithm.

Similar approaches to BigBWA are SEAL [70] and pBWA [75]. SEAL uses Pydoop [71], a Python implementation of the MapReduce programming model that runs on the top of Hadoop. It allows users to write their programs in Python, calling BWA methods by means of a wrapper. As we will show in the next section, using Pydoop introduces an overhead as compared to using JNI. pBWA uses a standard parallel programming paradigm as MPI to parallelize BWA. Unlike BigBWA, pBWA lacks fault tolerant mechanisms. There are another important differences between these tools and BigBWA. First, SEAL and pBWA only work with a particular modified version of BWA, while BigBWA works directly with the original BWA implementation. Therefore, no modifications to the BWA source code is required by BigBWA, keeping the compatibility with future and legacy BWA versions. Second, both SEAL and pBWA are based on BWA 0.5 version, which does not include the new BWA-MEM algorithm. Therefore, to the best of our knowledge, BigBWA is the first tool to handle the parallelization of the BWA-MEM algorithm using Big Data technologies.

BWA has its own parallel implementation, but it only supports shared memory machines. For this reason, scalability is limited by the number of threads (cores) available in one computing node. BigBWA, however, can be executed on clusters consisting of hundreds of computing nodes.

3.4. Discussion

Performance: BigBWA was tested using data from the 1000 Genomes Project [109] (see Table 3.1 for details). Measurements were performed on a 5-node AWS cluster with 16 cores per node (Intel Xeon E5-2670 at 2.5GHz CPUs), running Hadoop 2.6.0. Detailed information about the experimental setup is provided in the Supplementary Material. Performance results for BigBWA and the other evaluated tools only take into consideration the alignment process time, which was calculated as the average of five runs per data point after one warm-up execution. Table 3.2 shows a comparison with SEAL and pBWA for the BWA-backtrack algorithm. In this case, BigBWA clearly outperforms these tools, especially when the number of cores used is high. In this way, speedups of 36.4$\times$ were reached with respect to the sequential case (using the original BWA tool as reference). It can also be observed that the scalability of SEAL is worse, caused by the overhead introduced by Pydoop with respect to the use of JNI.

Performance of BWA-MEM is shown in Table 3.3. It was measured using only BWA (threaded version) and BigBWA, because SEAL and pBWA do not support this algorithm. We have also included results for a hybrid version that uses BigBWA in such a way that

each mapper processes the inputs using BWA with 2 threads. Results show that, with a small number of cores, BWA behaves slightly better than BigBWA. Note that BWA is limited to execute on just one cluster node and, therefore, we cannot provide results using more than 16 cores. Considering 16 cores, BigBWA is always the best solution but, due to the memory assigned per map task in our cluster configuration, only 13 concurrent tasks can be executed on one node. In this way, BigBWA always distributes the tasks between 2 nodes when using 16 cores. In addition, BigBWA shows good behavior in terms of scalability for all the datasets considered, executing up to 36.6× faster than the sequential case. Additional performance results are shown in the Supplementary Material.

Correctness: We verified the correctness of BigBWA by comparing its output file with the one generated by BWA. Differences range from 0.06% to 1% on uniquely mapped reads (mapping quality greater than zero), similarly to the differences shown by the threaded version of BWA with respect to the sequential case.

3.5. Supplementary material

3.5.1. BigBWA in more detail

BigBWA was designed as two independent software layers. The first one corresponds to the BWA software package, while the other is, strictly speaking, our tool. As we explain next, this design allows BigBWA to be version-agnostic regarding BWA, which assures the compatibility of BigBWA with any BWA version.

BigBWA relies on Hadoop as Big Data technology, which is the most successful open-source implementation of the MapReduce programming model introduced by Google [5]. Hadoop applications are typically developed in Java, but BWA is implemented in C. To overcome this issue, BigBWA takes advantage of the Java Native Interface (JNI) [77], which allows the incorporation of native code written in programming languages such as C and C++, as well as code written in Java. JNI is a mature and very commonly used technology. Actually, Hadoop itself uses JNI in some parts, since Hadoop core libraries are written in native code for performance reasons, and the programmer interacts with the Hadoop abstraction layer using Java.

To call the BWA methods using JNI from Hadoop Java code, it is necessary to create a shared library with the BWA source code (`libbwa.so`). No modifications to the original BWA source code are required to create this library, only the `-fPIC` flag must be included in

the BWA `Makefile` with the aim of generating position-independent code. In this way, the final BigBWA user does not need to worry about anything else about the JNI process.

Our JNI wrapper makes only calls to the `main` function of BWA. In this way, BigBWA parses all the input parameters required by BWA and, afterwards, passes the arguments to the BWA `main` function using the shared library `libbwa.so`. Using this simple approximation, no modifications to BWA are necessary and, as a consequence, future or legacy releases of BWA will be compatible with BigBWA. Only in case that BWA changes its input options/parameters among versions, BigBWA should modify the JNI wrapper. Therefore, BigBWA is completely agnostic of the internals of BWA.

Workflow

The complete BigBWA workflow can be summarized in the following steps:

1. Convert the input files to a Hadoop compatible format.
2. Copy the preprocessed input data to Hadoop HDFS.
3. Perform the alignment using BigBWA (Map and Reduce tasks).
4. Copy the output back to the local filesystem.

Step 1: Input files for BWA (and, by extension, for BigBWA) are in the *fastq* [110] format. The input data can be *single* (one entry file) or *paired* (two entry files). In the *fastq* format, each four lines from a file are considered as one *read*. On the other hand, Hadoop, by default, interprets each line of the input file as an individual entry for a mapper task. Therefore, it is necessary to preprocess the input files in such a way that one read (four lines in the original format) corresponds to one line in the Hadoop input file. Two Python scripts are provided with the BigBWA source code to preprocess the input *fastq* files:

- `Fq2FqBigData.py` - This script is used to adapt the input to run BigBWA with single reads. It takes as input one *fastq* file and produces the equivalent file with one read per line.
- `Fq2FqBigDataPaired.py` - This script is used to adapt the input to run BigBWA with paired reads. It takes as input two *fastq* files and produces one equivalent file with one paired read per line.

Step 2: Afterwards the preprocessed input files should be copied to Hadoop HDFS. This can be easily done using the CLI provided by Hadoop.

Step 3: BigBWA divides the computation into Map and Reduce phases. In the Map phase, BigBWA splits the reads into subsets, mapping each subset to a mapper process (data is read using HDFS). Each mapper is responsible for applying the considered BWA algorithm using as input the reads assigned by BigBWA. Note that mappers call BWA using the JNI wrapper commented above. Mappers are processed concurrently, speeding up the alignment process. In case any of the mappers fails, BigBWA would automatically launch another identical mapper process to replace the faulty one. At the end, BigBWA generates one output file per mapper. In the reducer phase those files are merged into one unique solution. Note that this phase is optional in such a way that users could choose to skip the reducer stage. In that case one output file per mapper would be generated.

Step 4: Copy back the output file/files from HDFS to the local filesytem, using again the Haddop CLI comands.

A complete tutorial is provided in the BigBWA GitHub repository: `https://github.com/citiususc/BigBWA`

3.5.2. Experimental setup

In order to test BigBWA we set up a Hadoop virtual cluster in Amazon EC2 [111]. Amazon gives their users the possibility of running a wide variety of virtual machines in their EC2 infrastructure. In our case, the virtual cluster consists of 5 nodes of the `r3.4xlarge` instance type. This kind of EC2 instances has the following characteristics:

- CPU: Intel Xeon E5-2670 v2 (Ivy Bridge microarchitecture)
- Cores per node: 16
- RAM Memory per node: 122 GB
- Disk: Each node has a 500 GB SSD General Purpose disk

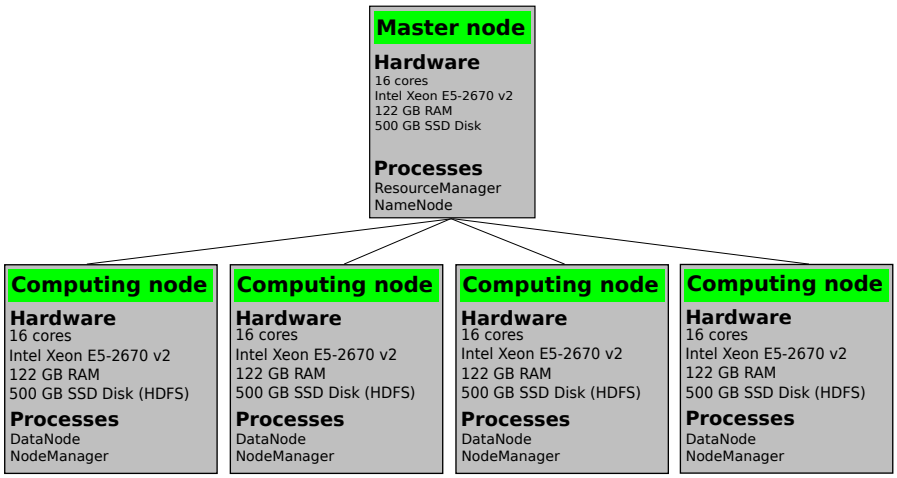


Figure 3.1: Structure of the Hadoop cluster used in the tests.

According to the specifications provided by Amazon [1], the network performance of the `r3.4xlarge` instances is "high". The theoretical bandwidth value is 1 Gbps, but experiments show that the actual bandwidth is between 100 Mbps and 1.86 Gbps[2].

An overview of the experimental setup running Hadoop 2.6.0 is shown in Figure 3.1.

Master node

One of these instances plays the role of Master node. This node is used to launch the Hadoop jobs and also to manage the Hadoop Distributed File System (HDFS). The Hadoop processes running on this node are the HDFS NameNode and the YARN Resource Manager. Additionally, the Master node allows the other (computing) nodes to access the reference genome index (5 GB) by means of NFS (Network File System).

Computing nodes

The remainder nodes are the Computing nodes, which perform the map and reduction operations. The cluster has a total of 64 computing cores (4 nodes) and 488 GB of memory

[1] http://aws.amazon.com/ec2/instance-types/
[2] http://www.aerospike.com/blog/boosting-amazon-ec2-network-for-high-throughput/

dedicated to computing. The Hadoop processes running on each of these nodes are the HDFS DataNode and the YARN NodeManager. Only the computing nodes are included in HDFS, so the filesystem has a total size of 2 TB. The block size specified by the Hadoop configuration files is 128 MB.

Memory requirements and scalability

One of the most important configuration parameters that influence the behavior of a Hadoop cluster is the memory assigned to the map and reduce tasks. In particular, those values are specified using `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` respectively. We have evaluated the memory requirements of the map/reduce tasks generated by BigBWA. Our experiments show that the average memory consumed per map/reduce task is 7.6 GB (ranging from 6 GB to 8.5 GB). For this reason both `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` are 9 GB in our cluster. Note that, for example, BWA uses a maximum of 5.5 GB and 8.8 GB for the sequential and 16-cores executions respectively. All these values were obtained using the datasets detailed in Table 3.4.

Another important observation is that there is no relation between the memory used by BigBWA and the input dataset size. Each map of BigBWA process one split of the input data, which consists of one or several HDFS blocks (128 MB per block in our cluster configuration). A map operates with one block at a time, processing line per line of the block in such a way that it is not necessary to keep all the data in memory. In this way, the memory consumed by a map task does not depend on the input data size. However, this is not the case of the reference index, which all the map tasks should keep in memory during their execution.

The fact that the memory needed for map (or reduce) tasks is assigned *a priori* could limit the scalability of a Hadoop application. For example, BigBWA runs on a cluster with 122 GB of memory per node, that is, 7.625 GB per core. As 9 GB are assigned per map/reduce task, BigBWA is not able to use all the 16 cores of the node. A maximum of 13 tasks can be concurrently executed on one node. Therefore, BigBWA distributes the tasks between 2 nodes for 16 cores executions, and among 3 and 5 nodes when using 32 and 64 cores respectively. Note that in the 64 cores case the master node is configured also as computing node. Add the master node as a computing node is very simple, and it only requires to change some Hadoop configuration parameters and reboot the YARN daemons.

All the datasets were extracted from the 1000 Genomes Project [109].

Tag	Name	Number of reads	Read length	Size
D1	NA12750/ERR000589	12×10^6	51 bp	3.9 GB
D2	HG00096/SRR062634	24.1×10^6	100 bp	13.4 GB
D3	150140/SRR642648	98.8×10^6	100 bp	54.7 GB

Table 3.4: Main characteristics of the input datasets.

Highlighted the best tool for a particular number of cores. For fair comparison with the other tools, BigBWA obtains these results using BWA version 0.5.10. Tool versions: pBWA 0.5.9 and SEAL 0.4.0.

Dataset	Tool	Pairs aligned/second						Speedup				
		Number of cores						Number of cores				
		1	4	8	16	32	64	4	8	16	32	64
D1	SEAL	135	359	707	901	1,802	3,509	2.7	5.2	6.7	13.4	26.0
	pBWA		**476**	791	1,130	2,174	3,922	**3.5**	5.9	8.4	16.1	29.1
	BigBWA		472	**840**	**1,299**	**2,353**	**4,444**	3.5	**6.2**	**9.6**	**17.3**	**33.0**
D2	SEAL	721	2,154	4,338	5,898	11,347	21,712	2.9	6.0	8.2	15.7	30.1
	pBWA		2,591	4,250	6,563	12,283	23,489	3.6	5.9	9.0	17.0	32.6
	BigBWA		**2,643**	**4,881**	**7,022**	**13,256**	**26,253**	**3.7**	**6.8**	**9.7**	**18.3**	**36.4**

Table 3.5: Comparison of the performance (pairs aligned/second) for the BWA-backtrack algorithm.

Highlighted the best tool for a particular number of cores. These results were obtained using BWA version 0.7.12.

Dataset	Tool	Pairs aligned/second						Speedup				
		Number of cores						Number of cores				
		1	4	8	16	32	64	4	8	16	32	64
D1	BWA-Threads	188	**725**	**1,399**	1,835	–	–	**3.9**	**7.4**	9.8	–	–
	BigBWA (hybrid)		676	1,325	1,695	2,941	5,556	3.6	7.0	9.0	15.7	29.6
	BigBWA		687	1,274	**2,532**	**4,444**	**6,667**	3.7	6.8	**13.4**	**23.5**	**35.5**
D2	BWA-Threads	1,557	**6,086**	**11,919**	16,131	–	–	**3.9**	**7.6**	10.4	–	–
	BigBWA (hybrid)		5,771	11,035	16,395	26,253	45,644	3.7	7.1	10.5	16.8	29.3
	BigBWA		5,813	10,711	**19,404**	**36,850**	**55,787**	3.7	6.9	**12.5**	**23.6**	**35.8**
D3	BWA-Threads	513	**2,016**	**4,035**	4,940	–	–	**3.9**	**7.9**	9.6	–	–
	BigBWA (hybrid)		1,987	3,820	**7,421**	8,983	15,361	3.9	7.4	**14.5**	17.5	29.9
	BigBWA		1,940	3,701	7,184	**13,711**	**18,755**	3.8	7.2	14.0	**26.7**	**36.6**

Table 3.6: Comparison of the performance (pairs aligned/second) for the BWA-MEM algorithm.

3.5.3. Additional performance results

Next we will include additional performance results to illustrate the benefits of BigBWA with respect to other existent tools. All the measurements were carried out using three datasets extracted from the 1000 Genomes Project [109] (see Table 3.4 for details). Performance results and comparisons with other similar software only take into consideration the alignment

process time (third step of the workflow in the case of BigBWA). Data values were calculated as the average of five runs.

First, we present the performance of BigBWA expressed in terms of the number of pairs aligned per second, which allows the readers to interpret more easily the scalability across both dataset and cluster size. Results are shown in Tables 3.5 and 3.6 for BWA-backtrack and BWA-MEM algorithms respectively (note that both tables correspond to the same results displayed in Tables 3.2 and 3.3 of the paper). According to the results for the BWA-backtrack algorithm, BigBWA obtain the best results overall and it is capable of aligning more than 26,000 pairs per second when using 64 cores. It can also be observed that, while speedups follow the same trend, the absolute numbers are completely different for both datasets. For instance, BigBWA processes about 2,300 or 13,200 pairs depending on the considered dataset when using 16 cores. The same behavior is observed for pBWA and SEAL. Therefore, performance is highly dependent on the considered dataset.

Results of the BWA-MEM algorithm are detailed in Table 3.6. Comparisons were performed using only BWA (threaded version) and BigBWA, because SEAL and pBWA do not support this algorithm. We have also included results for a hybrid version that uses BigBWA in such a way that each mapper processes the inputs using BWA with 2 threads. BWA behaves slightly better than BigBWA for small number of cores, but it is limited to execute on just one cluster node (that is, using a maximum of 16 cores). BigBWA clearly outperforms BWA from 16 cores on, aligning more than 55,000 pairs per second in the best case (`D2` dataset and 64 cores). For this algorithm we have also detected the same dependence of the performance with the dataset. In addition, larger data size is not equivalent to higher performance. In this way, best performance overall is always measured for `D2`, which is the medium size dataset, independently of the considered tool.

Finally, Figure 3.2 summarizes the average speedups using different number of cores for all the analyzed tools and algorithms with respect to BWA (sequential execution). BigBWA shows the best scalability among all the considered tools for both algorithms. We must highlight speedups of 34.7× and 36× with 64 cores for BWA-backtrack and BWA-MEM algorithms respectively. The efficiency of BigBWA, measured as the speedup and number of cores ratio, ranges from 0.54 (BWA-backtrack and 64 cores) to 0.93 (BWA-MEM and 4 cores).

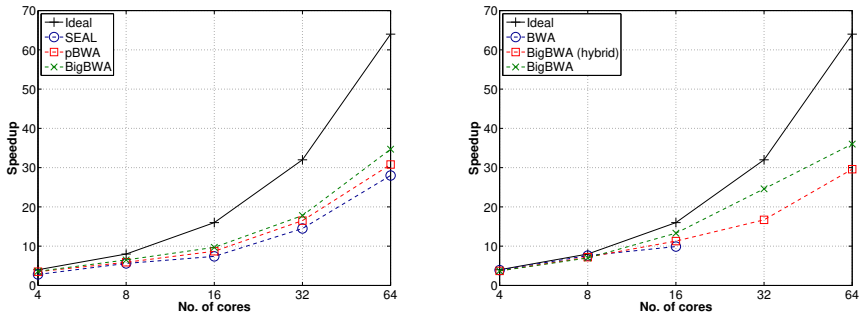


Figure 3.2: Average speedups for BWA-backtrack (left) and BWA-MEM (right) algorithms.

3.5.4. Related work

In addition to pBWA [71] and SEAL [70], which have been considered in the paper, we can find in the literature another interesting tools based on BWA. For example, BarraCUDA [112] takes advantage of the computing power of the GPUs to improve the performance of BWA using CUDA. This tool requires the modification of the BWT (Burrows Wheeler Transform) alignment core of BWA to exploit the massive parallelism of GPUs. Unlike BigBWA which supports all the algorithms included in BWA (that is, BWA-bactrack, BWA-SW and BWA-MEM), BarraCUDA only includes an implementation of the BWA-backtrack algorithm for short reads. It shows improvements up to 2× with respect to the threaded version of BWA. It is worth to mention that due to some changes in the BWT data structure of most recent versions of BWA, BarraCUDA is only compatible with BWTs generated with BWA versions 0.5.x. Another important aligners (not based on BWA) that make use of GPUs are: CUSHAW [72], SOAP3 [67] and SOAP3-dp [68].

Some researchers has focused on the new Intel Many Integrated Core (MIC) coprocessor technology. For example, in a recent work the authors introduced mBWA [113], which is an implementation of the BWA-backtrack algorithm for the Intel Xeon Phi coprocessor. mBWA allows to perform simultaneously the alignment process in both CPU and coprocessor, reaching speedups of 5× with respect to BWA. Another solution for the MIC coprocessors can be found in [114]. Another aligner that takes advantage of the MIC architecture is MICA [115]. Authors claim that it is 5× faster than threaded BWA using 6 cores. Note that, unlike BigBWA, this tool is not based on BWA.

CHAPTER 4

SPARKBWA: SPEEDING UP THE ALIGNMENT OF HIGH-THROUGHPUT DNA SEQUENCING DATA

Following is a reproduction of an article of which the author of this thesis is a main contributor. This is a verbatim reproduction, and the original can be found online under the following DOI: *10.1371/journal.pone.0155461*, or with this information:

J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "SparkBWA: speeding up the alignment of high-throughput DNA sequencing data," *PloS one*, vol. 11, no. 5, p. e0155461, 2016

4.1. Abstract

Next-generation sequencing (NGS) technologies have led to a huge amount of genomic data that need to be analyzed and interpreted. This fact has a huge impact on the DNA sequence alignment process, which nowadays requires the mapping of billions of small DNA sequences onto a reference genome. In this way, sequence alignment remains the most time-consuming stage in the sequence analysis workflow. To deal with this issue, state of the art aligners take advantage of parallelization strategies. However, the existent solutions show limited scalability and have a complex implementation. In this work we introduce SparkBWA,

a new tool that exploits the capabilities of a big data technology as Spark to boost the performance of one of the most widely adopted aligner, the Burrows-Wheeler Aligner (BWA). The design of SparkBWA uses two independent software layers in such a way that no modifications to the original BWA source code are required, which assures its compatibility with any BWA version (future or legacy). SparkBWA is evaluated in different scenarios showing noticeable results in terms of performance and scalability. A comparison to other parallel BWA-based aligners validates the benefits of our approach. Finally, an intuitive and flexible API is provided to NGS professionals in order to facilitate the acceptance and adoption of the new tool. The source code of the software described in this paper is publicly available at `https://github.com/citiususc/SparkBWA`, with a GPL3 license.

4.2. Introduction

The history of modern DNA sequencing starts more than thirty-five years ago. These years have seen amazing growth in DNA sequencing capacity and speed, especially after the appearance of next-generation sequencing (NGS) and massive parallel sequencing in general. NGS has led to an unparalleled explosion in the amount of sequencing data available. For instance, new sequencing technologies, such as Illumina HiSeqX™ Ten, generate up to 6 billion sequence reads per run. Mapping these data onto a reference genome is often the first step in the sequence analysis workflow. This process is very time-consuming and, although state-of-art aligners were developed to efficiently deal with large amount of DNA sequences, the alignment process still remains a bottleneck in bioinformatics analyses. In addition, NGS platforms are evolving very quickly, pushing the sequencing capacity to unprecedented levels.

To address this challenge we propose to take advantage of parallel architectures using big data technologies in order to boost performance and improve scalability of the sequence aligners. In this way, it will be possible to process huge amounts of sequencing data within a reasonable time. In particular, Apache Spark [13] has been considered as the big data framework in this work. Spark is a cluster computing framework which supports both in-memory and on-disk computations in a fault tolerant manner using distributed memory abstractions known as Resilient Distributed Datasets (RDDs). An RDD can be explicitly cached in memory across cluster nodes and reused in multiple MapReduce-like parallel operations.

In this paper we introduce SparkBWA, a new tool that integrates the Burrows-Wheeler aligner (BWA) [61] into the Spark framework. BWA is one of the most widely used align-

ment tools for mapping sequence reads to a large reference genome. It consists of three different algorithms for aligning short reads. SparkBWA was designed to meet three requirements. First, SparkBWA should outperform BWA and other BWA-based aligners both in terms of performance and scalability. Note that BWA has its own parallel implementation for shared-memory systems. The second requirement is related to keep the compatibility of SparkBWA with future and legacy versions of BWA. Since BWA is constantly evolving to include new functionalities and algorithms, it is important for SparkBWA to be agnostic regarding the BWA version. This is an important difference with respect to other existent tools based on BWA, which require modifications of the BWA source code. Finally, NGS professionals demand solutions to perform sequence alignments efficiently in such a way that the implementation details are completely hidden to them. For this reason SparkBWA provides a simple and flexible API to handle all the aspects related to the alignment process. In this way, bioinformaticians only need to focus on the scientific problem to deal with.

SparkBWA has been evaluated both in terms of performance and memory consumption, and a thorough comparison between SparkBWA and several state-of-art BWA-based aligners is also provided. Those tools take advantage of different parallel approaches as Pthreads, MPI, and Hadoop to improve the performance of BWA. Performance results demonstrate the benefits of our proposal.

This work is structured as follows: Section 4.3 explains the background of the paper. Section 4.4 discusses the related work. Section 4.5 details the design of SparkBWA and introduces its API. Section 4.6 presents the experiments carried out to evaluate the behavior and performance of our proposal together with a comparison to other BWA-based tools. Finally, the main conclusions derived from the work are explained in Section 4.7.

4.3. Background

4.3.1. MapReduce programming model

MapReduce [4] is a programming model introduced by Google for processing and generating large data sets on a huge number of computing nodes. A MapReduce program execution is divided into two phases: *map* and *reduce*. In this model, the input and output of a MapReduce computation is a list of key-value pairs. Users only need to focus on implementing map and reduce functions. In the map phase, map workers take as input a list of key-value pairs and generate a set of intermediate output key-value pairs, which are stored in the intermediate

storage (i.e., files or in-memory buffers). The reduce function processes each intermediate key and its associated list of values to produce a final dataset of key-value pairs. In this way, map workers achieve data parallelism, while reduce workers perform parallel reduction. Note that parallelization, resource management, fault tolerance and other related issues are handled by the MapReduce runtime.

Apache Hadoop [6] is the most successful open-source implementation of the MapReduce programming model. Hadoop consists, basically, of three layers: a data storage layer (HDFS – Hadoop Distributed File System [23]), a resource manager layer (YARN – Yet Another Resource Negociator [24]), and a data processing layer (Hadoop MapReduce Framework). HDFS is a block-oriented file system based on the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. For this reason, Hadoop shows good performance with embarrassingly parallel applications requiring a single MapReduce execution (assuming intermediate results between map and reduce phases are not huge), and even for applications requiring a small number of sequential MapReduce executions [34]. Note that Hadoop can also efficiently handle jobs composed by one or more map functions by chaining several mappers followed by a reducer function and, optionally, zero or more map functions, saving the disk I/O cost between map phases. For more complex workflows, solutions as Apache Oozie [35] or Cascading [36], among others, should be used.

The main disadvantage of these workflow managers is the loss of performance when HDFS has to be used to store intermediate data. For example, an iterative algorithm can be expressed as a sequence of multiple MapReduce jobs. Since different MapReduce jobs cannot shared data directly, intermediate results have to be written to disk and read again from HDFS at the beginning of the next iteration, with the consequent reduction in performance. It is worth noting that even each iteration of the algorithm could consist of one or several MapReduce executions. In this case, the degradation in terms of performance is even more noticeable.

4.3.2. Apache Spark

Apache Spark is a cluster computing framework designed to overcome the Hadoop limitations in order to support iterative jobs and interactive analytics, originally developed at University of California, Berkeley [13], now managed under the umbrella of the Apache Software Foundation. Spark uses a master/slave architecture with one central coordinator (driver) and many distributed workers (executors). It supports both in-memory and on-disk compu-

tations in a fault tolerant manner by introducing the idea of Resilient Distributed Datasets (RDDs) [39]. An RDD represents a read-only collection of objects partitioned across the cluster nodes that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. By using RDDs, programmers can perform iterative operations on their data without writing intermediary results to disk. In this way, Spark is well-suited, for example, to machine learning algorithms.

RDDs can be created by distributing a collection of objects (e.g., a list or set) or by loading an external dataset from any storage source supported by Hadoop, including the local file system, HDFS, Cassandra [116], HBase [29], Parquet [117], etc. On created RDDs, Spark supports two types of parallel operations: transformations and actions. Transformations are operations on RDDs that return a new RDD, such as map, filter, join, groupByKey, etc. The resulting RDD will be stored in memory by default, but Spark also supports the option of writing RDDs to disk whenever necessary. On the other hand, actions are operations that kick off a computation, returning a result to the driver program or writing it to storage. Examples are collect, count, take, etc. Note that transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.

A Spark application, at a high level, consists of a *driver* program which contains the application's main function and defines RDDs on the cluster, then applies transformations and actions to them. A Spark program implicitly creates, from defined transformations and actions over RDDs, a logical directed acyclic graph (DAG) of operations, which is converted by the driver into a physical execution plan. This plan is then optimized, e.g., merging several map transformations, and individual tasks are bundled up and prepared to be sent to the cluster. The driver connects to the cluster through a SparkContext. An *executor* or worker process is in charge of effectively running the tasks on each node of the cluster.

Apache Spark provides both Python and Scala interactive shells, which let the user interact with data that is distributed on disk or in memory across many machines. Apart from running interactively, Spark can also be linked into applications in either Java, Scala, or Python. Finally, we must highlight that Spark can run in local mode, in standalone mode on a cluster, or using a cluster manager such as Mesos [25] or YARN [24].

```
1 @ERR000589.41 EAS139_45:5:1:2:111/1
2 CTTTCCTCCCTGCTTTCCTGGCCCCACCATTTCCAGGGAACATCTTGTCAT
3 +
4 3IIIIIIIIIIIII>1IIIFF9BG08E00I%IG+&?(4)%00646.C1#&(
5 @ERR000589.42 EAS139_45:5:1:2:1293/1
6 AGTTGTTAAAATCCAAGCCAATTAAGATAGTCTTATCTTTTTAAAAGAAAT
7 +
8 IIIIIGII.AIIII=?I9G-/II=+I=4?761BA2C9I+5A711+&>1$/I
```

Figure 4.1: FASTQ file format example.

4.3.3. Burrows-Wheeler aligner (BWA)

Burrows-Wheeler aligner (BWA) is a very popular open-source software for mapping sequence reads to a large reference genome. In particular, it consists of three different algorithms: BWA-backtrack [61], BWA-SW [62] and BWA-MEM [63]. The first algorithm is designed for short Illumina sequence reads up to 100bp (base pairs), while the others are focused on longer reads. BWA-MEM, which is the latest, is preferred over BWA-SW for 70bp or longer reads as it is faster and more accurate. In addition, BWA-MEM has shown better performance than other several state-of-art read aligners for mapping 100bp or longer reads.

As we have previously noted, sequence alignment is a very time-consuming process. For this reason BWA has its own parallel implementation, but it only supports shared memory machines. Therefore, scalability is limited by the number of threads (cores) and memory available in just one computing node.

Although BWA can read unaligned BAM [118] files, it typically accepts FASTQ format [110] as input, which is one of the most common output formats for raw sequence reads. It is a plain text format in such a way that every four lines describe a sequence or read. An example including two reads is shown in Figure 4.1. The information provided per read is: identifier (first line), sequence (second line), and the quality score of the read (fourth line). An extra field, represented by symbol '+', is used as separator between the data and the quality information (third line). BWA is able to use single-end reads (one input FASTQ file) and paired-end reads (two input FASTQ files). When considering paired-end reads, two sequences corresponding to both ends of the same DNA fragment are available. Both reads are included in different input files using the same identifier and in the same relative location within the files. In this way, considering our example, the corresponding pair of sequence #2 will be located in line 5 of the other input file. On the other hand, the output of BWA is a SAM (Sequence Alignment/Map) [118] file, which is the standard format for storing read align-

ments against reference sequences. This SAM file will be further required, for example, for performing variant discovery analysis.

4.4. Related Work

We can find in the literature several interesting tools based on the Burrows-Wheeler aligner which exploit parallel and distributed architectures to increase the BWA performance. Some of these works are focused on big data technologies like SparkBWA, but they are all based on Hadoop. Examples are BigBWA [76], Halvade [69] and SEAL [70]. BigBWA is a recent sequence alignment tool developed by the authors which shows good performance and scalability results with respect to other BWA-based approaches. Its main advantage is that it does not require any modification of the original BWA source code. This characteristic is shared by SparkBWA in such a way that both tools keep the compatibility with future and legacy BWA versions.

SEAL uses Pydoop [71], a Python implementation of the MapReduce programming model that runs on the top of Hadoop. It allows users to write their programs in Python, calling BWA methods by means of a wrapper. SEAL only works with a particular modified version of BWA. Since SEAL is based on BWA version 0.5, it does not support the new BWA-MEM algorithm for longer reads.

Halvade is also based on Hadoop. It includes a variant detection phase which is the next stage after the sequence alignment in the DNA sequencing workflow. Halvade calls BWA from the mappers as an external process which may cause timeouts during the Hadoop execution if the task timeout parameter is not adequately configured. Therefore, *a priori* knowledge about the execution time of the application is required. Note that setting the timeout parameter to high values causes problems in the detection of actual timeouts, which reduces the efficiency of the fault tolerance mechanisms of Hadoop. To overcome this issue, as it is explained in further sections, SparkBWA uses Java Native Interface (JNI) to call the BWA methods.

Another approach is applying standard parallel programming paradigms to BWA. For instance, pBWA [75] uses MPI to parallelize BWA in order to carry out the alignments on a cluster. We must highlight that pBWA lacks fault tolerant mechanisms in contrast to SparkBWA. In addition, pBWA, as well as SEAL, does not support the BWA-MEM algorithm.

Several solutions try to take advantage of the computing power of the GPUs to improve

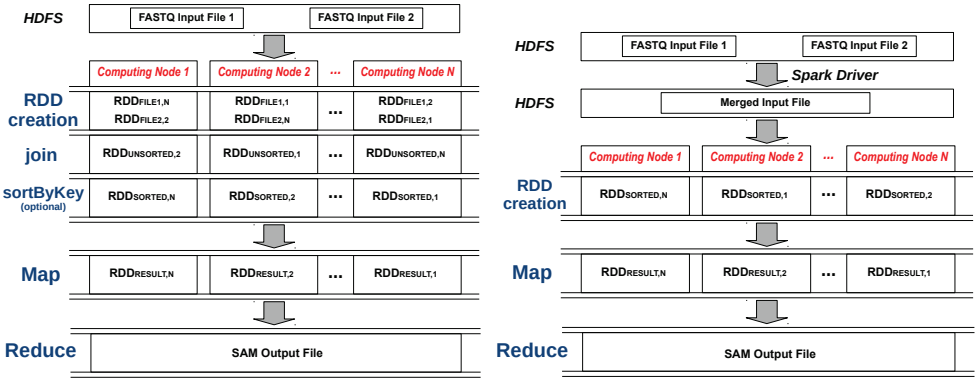


Figure 4.2: SparkBWA workflow for paired-end reads using (a) Join and (b) SortHDFS approaches.

the performance of BWA. This is the case of BarraCUDA [112], which is based on the CUDA programming model. It requires the modification of the BWT (Burrows Wheeler Transform) alignment core of BWA to exploit the massive parallelism of GPUs. Unlike SparkBWA which supports all the algorithms included in BWA, BarraCUDA only supports the BWA-backtrack algorithm for short reads. It shows improvements up to 2× with respect to the threaded version of BWA. It is worth to mention that due to some changes in the BWT data structure of most recent versions of BWA, BarraCUDA is only compatible with BWTs generated with BWA versions 0.5.x. Other important sequence aligners (not based on BWA) that make use of GPUs are CUSHAW [72], SOAP3 [67] and SOAP3-dp [68].

Some researchers have focused on speeding up the alignment process using the new Intel Xeon Phi coprocessor (Intel Many Integrated Core architecture - MIC). For example, mBWA [113], which is based on BWA, implements the BWA-backtrack algorithm for the Xeon Phi coprocessor. mBWA allows to use concurrently both host CPU and coprocessor in order to perform the alignment, reaching speedups of 5× with respect to BWA. Another solution for the MIC coprocessors can be found in [114]. A third aligner that takes advantage of the MIC architecture is MICA [115]. Authors claim that it is 5× faster than threaded BWA using 6 cores. Note that, unlike SparkBWA, this tool is not based on BWA.

Another researchers exploit fine-grain parallelism in FPGAs (Field Programmable Gate Arrays) to increase the performance of several short-read aligners including some based on

BWT [119, 120, 121].

Finally, a recent work uses Spark to increase the performance of one of the best well-known alignment algorithms, the Smith-Waterman algorithm [122]. Performance results demonstrate the potential of Spark as framework for this type of applications.

4.5. SparkBWA

This section introduces a new tool called SparkBWA, which integrates the Burrows-Wheeler aligner into the Spark framework. As stated in the Introduction, SparkBWA was designed with the following three objectives in mind:

- It should boost BWA and other aligners based on BWA in terms of performance and scalability.
- It should be version-agnostic regarding BWA, which assures its compatibility with future or legacy BWA versions.
- An intuitive and flexible API should be provided to NGS professionals with the aim of facilitating the acceptance and adoption of the new tool.

Next, a detailed description of the design and implementation of SparkBWA is provided, together with the specification of the high-level API.

4.5.1. System design

SparkBWA workflow consists of three main stages: RDDs creation, map, and reduce phases. In the first phase input data are prepared to feed the map phase where the alignment process is, strictly speaking, carried out. In particular, RDDs are created from the FASTQ input files, which are stored using HDFS. Note that, in this work, we assume HDFS as distributed file system. In this way, data is distributed across the computing nodes so it can be processed in parallel in the map phase. The read identifier in the FASTQ file format is used as *key* in the RDDs (see the example of Figure 4.1). In this way, key-value pairs generated from an input file have the following appearance <*read_id, read_content*>, where *read_content* contains all the information of the corresponding sequence with *read_id* identifier. These RDDs will be used afterwards in the map phase. This approach works properly when considering single-end reads, that is, when there is only one FASTQ input file.

However, SparkBWA should also support paired-end reads. In that case, two RDDs will be created, one per input file, and distributed among the nodes. Spark distributes RDDs in such a way that is not guaranteed that the *i*-th data split (partition) of both RDDs will be processed by the same mapper. In this way, a mapper cannot process paired-end reads since they are always located in the same *i*-th data partition of both RDDs. This behavior can be observed in the RDD creation stage of the example displayed in Figure 4.2(a). Two solutions are proposed in order to overcome this issue:

- **Join**: This approach is based on using the Spark *join* operation, which is a transformation that merges two RDDs together by grouping elements with the same key. This solution is illustrated in Figure 4.2(a). Since the key is the same for paired reads in both input files, the result after the join operation will be an unique RDD with the format: *<read_id, Tuple<read_content1, read_content2>>* ($RDD_{UNSORTED}$ in the example). The resulting RDD after the join operation does not preserve the previous order of the reads from the FASTQ files. This is not a problem because mappers will process the paired-end reads independently from each other. However, Spark provides the *sortByKey* transformation to sort RDD records according to its key. In the example, the new RDD created after applying this operation is RDD_{SORTED}. We must highlight that the sortByKey operation is expensive in terms of memory consumption. For this reason this step is optional in the SparkBWA dataflow and users should enable it specifically, if they want to get a sorted output.

- **SortHDFS**: A new approach is presented in order to avoid the *join* and *sortByKey* operations (see Figure 4.2(b)). This solution can be considered as a preprocessing stage which requires reading and writing to/from HDFS. In this way, FASTQ input files are accessed directly by using the HDFS Hadoop library from the Spark driver program. Paired-end reads (that is, those with the same identifier in the two files) are merged into one record in a new HDFS file. As BWA requires to distinguish between both sequences in the pair, a separator string is used to facilitate the subsequent parsing process in the mappers. Afterwards, an RDD is created from the new file (RDD_{SORTED} in the figure). In this way, key-value pairs have the following format *<read_id, merged_content>*.

  This solution performs several time consuming I/O operations, but saves a lot of memory in comparison to the join & sortByKey approach as we illustrate in Section 4.6.

Function	Default	Console argument	Description
`setUseReducer(boolean)`	False	`-r`	Use a reducer to generate one output SAM file.
`setPartitionNumber(int)`	Auto	none |`-partitions <num>`	By default, data is split into pieces of HDFS block size. Otherwise, input data is split into `num` partitions.
`setSortFastqReads(int)`	Join	none |`-sort` |`-sorthdfs`	Set the RDDs creation approach for paired-end reads: Join (0), Join & sortByKey (1) or SortHDFS (2).
`setNumThreads(int)`	1	`-threads <num>`	If `num` $>$ 1, hybrid parallelism mode is enabled in such a way that each map process is executed using *num* threads.
`setAlgorithm(int)`	BWA-MEM	`-mem` |`-aln` |`-bwasw`	Set the alignment algorithm: BWA-MEM (0), BWA-backtrack (1), BWA-SW (2)
`setPairedReads(boolean)`	Paired	`-paired` |`-single`	Use single-end (one FASTQ input file) or paired-end reads (two FASTQ input files).
`setIndexPath(string)`	–	`-index <prefix>`	Set the path to the reference genome (mandatory option).
`setInputPath(string)`	–	Positional	Set the path (in HDFS) to the FASTQ input file (mandatory option for single-end and paired-end reads).
`setInputPath2(string)`	–	Positional	Set the path (in HDFS) to the second FASTQ input file (mandatory option for paired-end reads).
`setOutputPath(string)`	–	Positional	Set the location (in HDFS) where the output SAM file/s will be stored.

Table 4.1: API methods and console arguments to set the SparkBWA options

Once RDDs are available, the map phase starts. Mappers will apply the sequence alignment algorithm from BWA on the RDDs. However, calling BWA from Spark is not straightforward as BWA source code is written in C language and Spark only allows to run code in Scala, Java or Python. To overcome this issue SparkBWA takes advantage of the Java Native Interface (JNI), which allows the incorporation of native code written in languages as C and C++ as well as Java code.

The map phase was designed using two independent software layers. The first one corre-

sponds to the BWA software package, while the other is responsible to process RDDs, pass the input data to the BWA layer and collect the partial results from the map workers. We must highlight that mappers only perform calls to the BWA main function by means of JNI. This design avoids any modification of the original BWA source code, which assures the compatibility of SparkBWA with future or legacy BWA versions. In this way, our tool is version-agnostic regarding BWA. Note that this approach is similar to the one adopted in the BigBWA tool [76].

Another advantage of the two-layers design is that the alignment process could be performed using two levels of parallelism. The first level corresponds to the map processes distributed across the cluster. In the second level each individual map process is parallelized using several threads, taking advantage of the BWA parallel implementation for shared memory machines. We refer to this mode of operation as *hybrid mode*. This mode can be enabled by the user through the SparkBWA API.

On the other hand, BWA uses a reference genome as input in addition to the FASTQ files. All mappers require the complete reference genome, so it has to be shared among all computing nodes using NFS or stored locally in the same location of all the nodes (e.g., using Spark broadcast variables).

Once the map phase is complete, SparkBWA creates one output SAM file in HDFS per launched map process. Finally, users could merge all the outputs into one file choosing to execute an additional reduce phase.

4.5.2. SparkBWA API

One of the requirements of SparkBWA is to provide bioinformaticians an easy and powerful way to perform sequence alignments using a big data technology as Apache Spark. With this goal in mind a basic API is provided. It allows NGS professionals to focus only in the scientific problem, while design and implementation details of SparkBWA are completely transparent to them.

SparkBWA can be used from the Spark shell (Scala) or console. Table 4.1 summarizes the API methods to set the SparkBWA options in the shell together with their corresponding console arguments. For example, it is possible to choose the number of data partitions, how RDDs are created, or the number of threads used per mapper (hybrid mode).

1. *Spark Shell*: Spark comes with an interactive shell that provides a simple way to learn the

```
1  scala> var options = new BwaOptions();
2
3  scala> options.setInputPath("ERR000589_1.filt.fastq");
4  scala> options.setInputPath2("ERR000589_2.filt.fastq");
5
6  scala> options.setOutputPath("OutputSparkBWA.sam");
7  scala> options.setIndexPath("/opt/HumanBase/hg38");
8
9  scala> var newBwa = new BwaInterpreter(options, sc);
10 scala> var bwaRDD = newBwa.getDataRDD(); # Optional
11 scala> newBwa.runAlignment();
```

Figure 4.3: Example running SparkBWA from the Spark Shell (Scala).

Spark API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. Current SparkBWA version only supports the Scala shell.

An example of how to perform an alignment using SparkBWA from the Spark shell is illustrated in Figure 4.3. First, the user should create a `BwaOptions` object to specify the options desired in order to execute SparkBWA (line 1). In this example only the mandatory options are set (lines 3 – 7). Refer to Table 4.1 for additional options.

Once the options are specified, a new `BwaInterpreter` should be created (line 9). At that moment RDDs are created from the input files according to the implementation detailed previously in Section 4.5.1. It is worth to mention that the RDDs creation is lazy evaluated, which means that Spark will not begin to execute until an action is called. This action could be, for example, obtaining explicitly the input RDD using the `getDataRDD` method (line 10). This method is very useful in the sense that it allows the users to apply to the input RDDs all the transformations and actions that the Spark API provides in addition to user-defined functions. Note that using the `getDataRDD` method is not necessary to perform the sequence alignment with SparkBWA. Another action that triggers the RDDs creation is `runAlignment`, which will execute the complete SparkBWA workflow including the map and reduce phases (line 11).

2. *Console*: It is also possible to run SparkBWA from the console, that is, using the `spark-submit` command. An example is shown in Figure 4.4. `spark-submit` provides a variety of options that let the user control specific details about a particular run of an application (lines 2 – 6). In our case, the user also needs to pass as arguments the SparkBWA options

```
1  spark-submit
2  --class SparkBWA
3  --master yarn-client          # Connect to a YARN cluster
4  --num-executors 16            # Number of worker processes
5  --archives bwa.zip            # BWA library
6  SparkBWA.jar                  # SparkBWA tool
7  -partitions 16                # Data partitions
8  -index /opt/HumanBase/hg38    # Reference genome
9  ERR000589_1.filt.fastq        # Input file 1
10 ERR000589_2.filt.fastq        # Input file 2
11 OutputSparkBWA.sam            # Output file
```

Figure 4.4: Example running SparkBWA from the console.

Tag	Name	Number of reads	Read length (bp)	Size (GiB)
D1	NA12750/ERR000589	12×10^6	51	3.4
D2	HG00096/SRR062634	24.1×10^6	100	11.8
D3	150140/SRR642648	98.8×10^6	100	48.3

Table 4.2: Main characteristics of the input datasets from the 1000 Genomes Project.

to Spark (lines 7 – 11). All the flags supported by SparkBWA are detailed in Table 4.1.

Therefore, SparkBWA provides an easy and flexible interface in such way that users could perform a sequence alignment writing just a couple of lines of code in the Spark shell, or using the standard `spark-submit` tool from the console.

4.6. Evaluation

In this section SparkBWA is evaluated in terms of performance, scalability, and memory consumption. First, a complete description of the experimental setup is provided. Next, SparkBWA is analyzed in detail paying special attention to the creation of RDDs and its different modes of operation (regular and hybrid). Finally, in order to validate our proposal, a comparison to several BWA-based aligners is also provided.

4.6.1. Experimental Setup

SparkBWA was tested using data from the 1000 Genomes Project [109]. The main characteristics of the input datasets are shown in Table 4.2. Number of reads refers to the number

Algorithm	Tools	Parallelization Technology
BWA-backtrack	pBWA [75]	MPI
	SEAL [70]	Hadoop
	SparkBWA	Spark
BWA-MEM	BWA [63]	Pthreads
	BigBWA [76]	Hadoop
	Halvade [69]	Hadoop
	SparkBWA	Spark

Table 4.3: Algorithms and BWA-based aligners evaluated.

of sequences to be aligned to the reference genome. The read length is expressed in terms of the number of base pairs (bp).

As the alignment can be performed for single or paired-ended reads, it is needed to determine which one is going to be used during the evaluation. As the paired-ended DNA sequencing reads provide superior alignment across DNA regions containing repetitive sequences reads, it is the one that is considered in this work. In this way, each dataset consists of two FASTQ files.

Experiments were carried out on a six-node cluster. Each node consists of four AMD Opteron 6262HE processsors (4×16 cores) with 256 GiB of memory (i.e., 4 GiB per core). Nodes are connected through a 10GbE network. The Hadoop and Spark versions used are 2.7.1 and 1.5.2, respectively, running on a CentOS 6.7 platform. OpenMPI 4.4.7 was used in the experiments that require MPI. The cluster was configured assigning about 11 GiB of memory per YARN container (map and reduce processes) in such a way that a maximum of 22 containers per node can be executed concurrently. This memory configuration allows each SparkBWA container to execute one BWA process, including the memory required to store the reference genome index. Note that the master node in the cluster is also used as computing node.

The behavior of SparkBWA is compared to several state of the art BWA-based aligners. In particular, we have considered the tools detailed in Table 4.3. A brief description of these tools is provided in Section 4.4. pBWA and SEAL only support the BWA-backtrack algorithm because both are based on BWA version 0.5 (2009). For fair comparison with these tools, SparkBWA obtains its performance results for the BWA-backtrack algorithm also using BWA version 0.5. In the case of BWA-MEM, three different aligners are evaluated: BigBWA, Halvade and BWA (shared-memory threaded version). For the BWA-MEM performance evaluation, the latest available BWA version at the moment of writing the paper is used (version

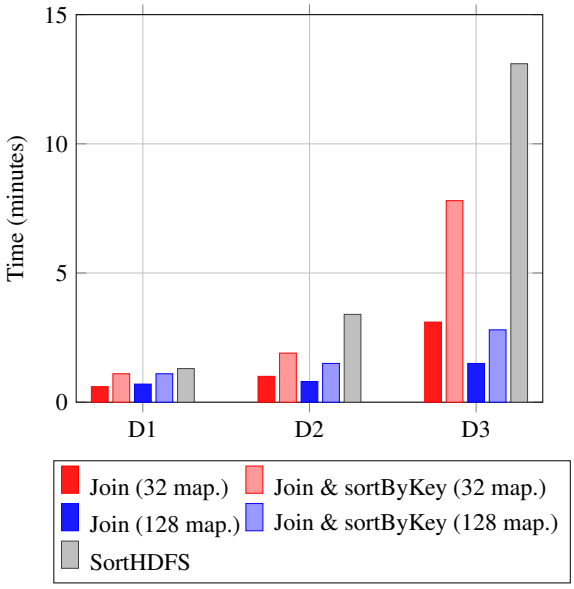


Figure 4.5: Overhead of the RDDs sorting operation considering different datasets.

0.7.12, December 2014). We must highlight that all the time results shown in this section were calculated as the average value (arithmetic mean) of twenty executions.

4.6.2. Performance Evaluation

RDDs creation

The first stage in the SparkBWA workflow is the creation of the RDDs, which can include a sorting phase (see Section 4.5.1). Two different approaches were considered to implement this phase: Join and SortHDFS. The first one is based on the Spark *join* operation, and includes an additional optional step to sort the input paired-end reads by key (*sortByKey* operation). The latter approach requires reading and writing to/from HDFS. As we pointed out previously, this solution can be considered as a preprocessing stage. Both solutions have been evaluated in terms of the overhead considering different datasets. Results are displayed in Figure 4.5.

The performance of the Join approach (with and without the *sortByKey* transformation) depends on the number of map processes, so this operation was evaluated using 32 and 128 mappers. As the number of mappers increases, the sorting time improves because the size of

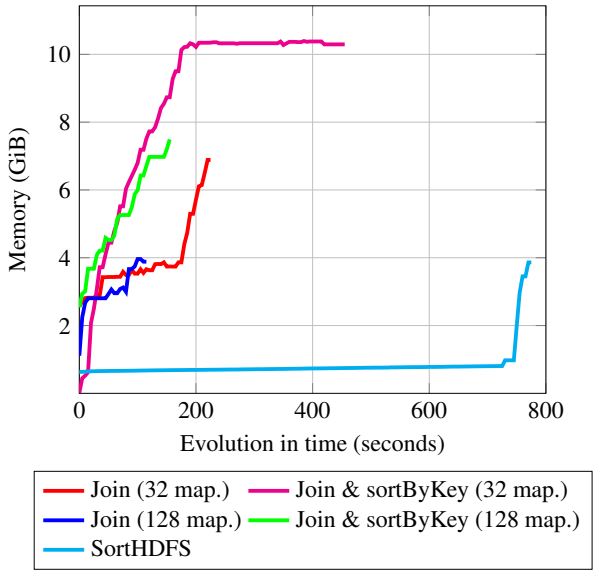


Figure 4.6: Memory consumed by SparkBWA during the RDDs sorting operation when considering dataset D3.

the data splits computed by each worker is smaller. This behavior was observed for all the datasets, especially when D3 is considered.

The overhead for all the approaches, as it was expected, increases with the size of the dataset. However, the increment rate is higher for SortHDFS. For example, sorting D3 is 10× slower than sorting D1, while the Join approach with and without *sortByKey* is at most only 5× and 7× slower respectively. Note that D3 is more than 14× bigger than D1 (see Table 4.2).

The Join approach is always better in terms of overhead, especially as the number of map processes increases. For example, sorting D3 takes only 1.5 minutes with 128 mappers (*join* only), which means a speedup of 8.7× with respect to SortHDFS. It can also be observed that sorting the RDDs by key consumes extra time. In particular, the overhead means on average doubling the time required by the sorting process when only the *join* transformation is performed.

On the other hand, speed is not the only parameter that should be taken into account when performing the RDDs sorting. In this way, memory consumption has also been analyzed. In order to illustrate the behavior of both sorting approaches we have considered D3 as dataset.

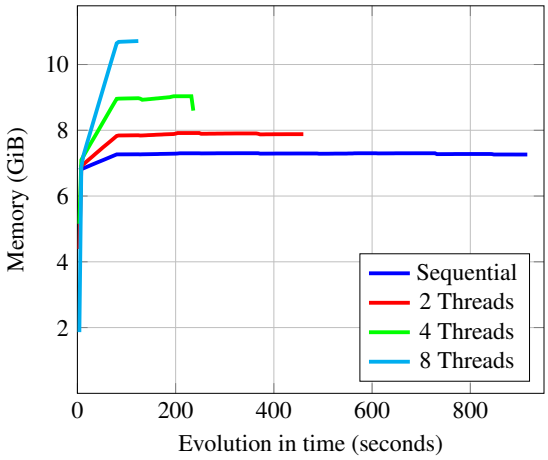


Figure 4.7: Memory consumed by a worker process executing the BWA-MEM algorithm with different threads.

Figure 4.6 shows the memory used by a map process during the sorting operation period.

According to the results, the Join approach always consumes more memory than SortHDFS. This is caused by the *join* and *sortByKey* Spark operations on the RDDs, which both are in-memory transformations. It is especially relevant the differences observed when the elements of the RDDs are sorted by key with respect to applying only the *join* operation. In this way, the *sortByKey* operation consumes about 3 GiB extra per mapper for this dataset, which means increasing more than 30% the memory required by SparkBWA in this phase. Note that when considering 32 workers the maximum memory available per container is reached. The memory used by 128 workers is lower because RDDs are split into smaller pieces with respect to considering 32 workers. On the other hand, SortHDFS requires a maximum of 4 GiB to preprocess the dataset in the example. In this way, SortHDFS is the best choice if the memory resources are limited or not enough to perform the Join operation (with or without *sortByKey*). Note that the overall behavior illustrated in Figure 4.6 agrees with the observations for the other datasets.

Hybrid mode

As stated in Section 4.5.1, the design of SparkBWA in two software layers allows to use several threads per worker in such a way that the alignment process is performed taking

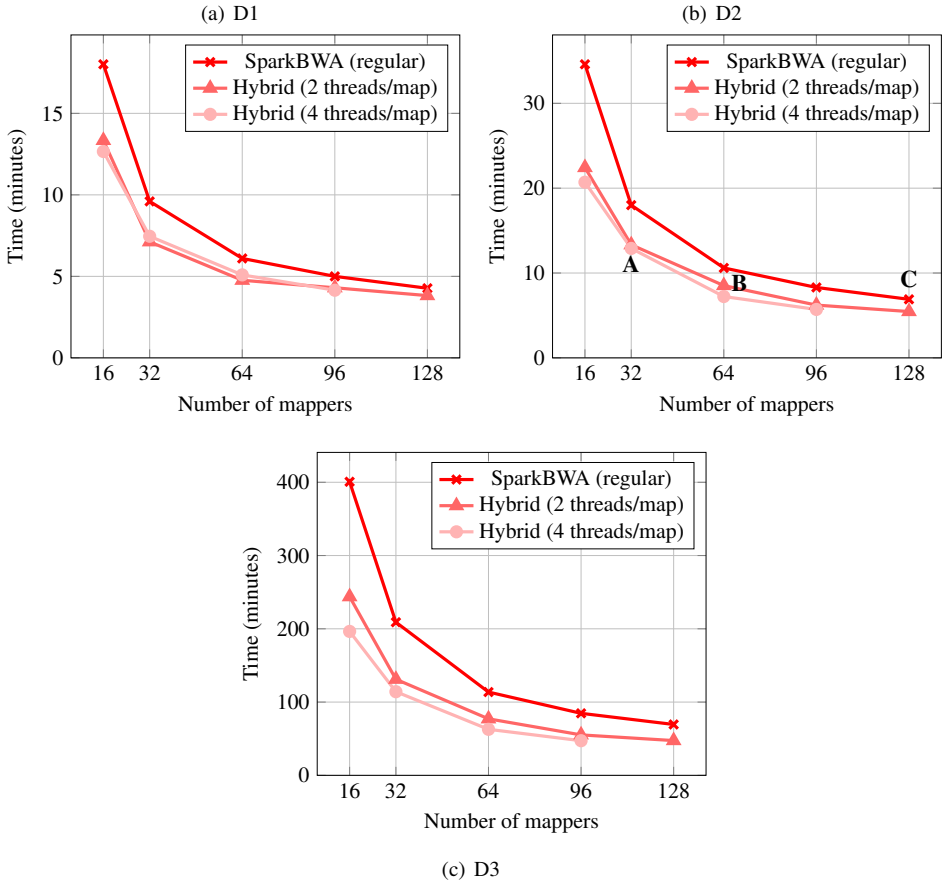


Figure 4.8: Execution times obtained by SparkBWA using regular and hybrid modes of operation for the BWA-MEM algorithm.

advantage of two levels of parallelism. In this way, SparkBWA has two modes of operation: regular and hybrid. The hybrid mode refers to using more than one thread per map process, while the regular behavior executes each mapper sequentially.

The memory used by each mapper when hybrid mode is enabled increases with the number of threads involved in the computation. However, since the index reference genome required by BWA is shared among threads, this increase is moderate. This behavior is illustrated in Figure 4.7, where BWA-MEM is executed using different number of threads with a small

split of D1 as input. It can be observed that the difference between the memory used by one SparkBWA mapper considering regular and hybrid mode with 8 threads is only 4 GiB. It means an increase of about 30% in the total memory consumed, while the threads per mapper grows by a factor of 8.

So, taking into account that our experimental platform allows 22 containers per node with 11 GiB of maximum memory, SparkBWA in hybrid mode for this example could use all the 64 cores in the node, e.g., running 16 mappers and 4 threads/mapper. This is not the case of the regular mode, which only allows to use a maximum of 22 cores of the node. Therefore, the hybrid mode can be very useful in scenarios where the computing nodes consist of a high number of cores but, due to memory restrictions, only a few of them can be used.

Next, we evaluate the performance of SparkBWA using both modes of operation. Experiments were conducted using the BWA-MEM algorithm and considering 2 and 4 threads per map process when hybrid mode is enabled. Performance results are shown in Figure 4.8 for all the datasets and using different number of mappers. There are no results for the 128 mappers with 4 threads/mapper case because it implies that 512 cores are necessary for an optimal execution, while our cluster only consists of 384 cores.

Several conclusions can be extracted from the performance results. SparkBWA shows a good scalability with the number of mappers, especially in the regular mode (that is, when each mapper is computed sequentially). Assuming the same number of mappers, more threads per mapper in the hybrid mode is only beneficial for the biggest dataset (D3). This behavior points out that the benefits of using more threads in the computations do not compensate the overhead caused by their synchronization.

On the other hand, considering the cores used in the computation (#threads $\times$ #mappers cores), we can observe that the regular mode performs better than the hybrid one. For instance, points A, B and C in Figure 4.8(b) were obtained using the same number of cores. SparkBWA in regular mode (point C) clearly outperforms the hybrid version. This behavior is observed in most of the cases. In this way, as we have indicated previously, SparkBWA hybrid mode should be the preferred option only in those cases where limitations in memory do not allow to use all the cores in each node.

Table 4.4 summarizes the results of SparkBWA in terms of performance for all the datasets. It shows the minimum time required by SparkBWA to perform the alignment on our hardware platform, the number of mappers used, the speed measured as the number of pairs aligned per second and also the corresponding speedup with respect to the sequential execution of BWA.

Dataset	Mode of operation	No. of mappers	Time (minutes)	Pairs aligned/s	Speedup
D1	regular	128	4.3	46,512	60×
	hybrid (2 th/map)	128	3.8	52,632	67.9×
	hybrid (4 th/map)	96	4.1	48,780	62.9×
D2	regular	128	6.9	58,213	71.9×
	hybrid (2 th/map)	128	5.5	73,030	90.2×
	hybrid (4 th/map)	96	5.7	70,468	87.0×
D3	regular	128	69.4	23,727	85.6×
	hybrid (2 th/map)	128	47.5	34,667	125.0×
	hybrid (4 th/map)	96	47.3	34,813	126.2×

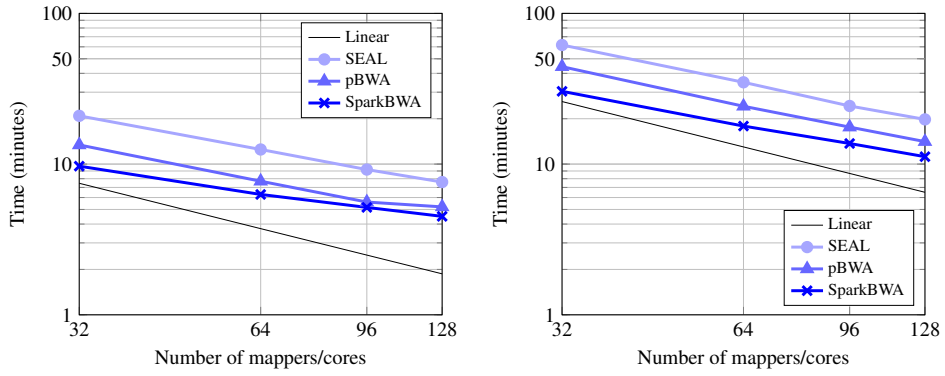
Table 4.4: Summary of the performance results of SparkBWA.

The sequential times are respectively 258, 496 and 5,940 minutes for D1, D2 and D3. In the particular case of D3 it means more than 4 days of computation. It is worth noting that using SparkBWA this time was reduced to less than an hour reaching speedups higher than 125×.

Finally, we verified the correctness of SparkBWA for regular and hybrid modes by comparing their output with the one generated by BWA (sequential version). We only found small differences in the mapping quality scores (mapq) on some uniquely mapped reads (i.e., reads with quality greater than zero). Therefore, the mapping coordinates are identical for all the cases considered. Differences affect from 0.06% to 1% of the total number of uniquely mapped reads. Small differences in the mapq scores are expected because the quality calculation depends on the insert size statistics, which are calculated on sample windows on the input stream of sequences. These sample windows are different for each read in BWA (sequential) and any other parallel implementation that splits the input into several pieces (SEAL, pBWA, Halvade, BWA-threaded version, SparkBWA, etc.). In this way, any parallel BWA-based aligner will obtain slightly different mapping quality scores with respect to the sequential version of BWA. For instance, SEAL reports differences on average in 0.5% of the uniquely mapped reads [70].

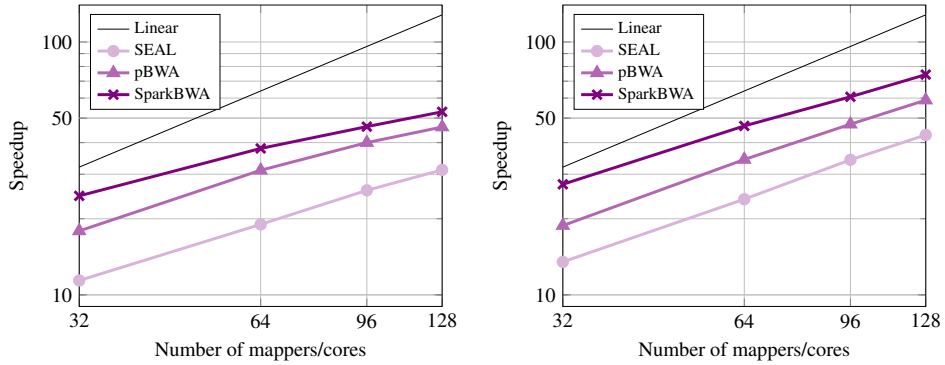
Comparison to other aligners

Next, a performance comparison among different BWA-based aligners and SparkBWA is shown. The evaluated tools are enumerated in Table 4.3 together with their corresponding parallelization technology. Some of them take advantage of classical parallel paradigms, as Pthreads or MPI, while the others are based on big data technologies as Hadoop. All the experiments were performed using SparkBWA in regular mode. For comparison purposes all



(a) D1 (b) D2

Figure 4.9: Execution times considering several BWA-based aligners running the BWA-backtrack algorithm (axes are in log scale).



(a) D1 (b) D2

Figure 4.10: Speedup considering several BWA-based aligners running the BWA-backtrack algorithm (axes are in log scale).

the graphs in this subsection include the corresponding results considering ideal speedup with respect to the sequential execution of BWA.

Two different algorithms for paired-end reads have been considered: BWA-backtrack and BWA-MEM. The evaluation of the BWA-backtrack algorithm was performed using the following aligners: pBWA, SEAL and SparkBWA. When paired reads are used as input data,

BWA-backtrack consists of three phases. First, the sequence alignment must be performed for one of the input FASTQ files. Afterwards, the same action is applied to the other input file. Finally, a conversion to the SAM output format is performed using the results of the previous stages. SparkBWA and SEAL take care of the whole workflow in such a way that it is completely transparent to the user. Note that SEAL requires a preprocessing stage to prepare the input files, so this extra time was included in the measurements. On the other hand, pBWA requires to perform each phase of the BWA-backtrack algorithm independently despite they are executed in parallel. In this way, pBWA times were calculated as the sum of each phase time. No preprocessing is performed by pBWA.

As BWA-backtrack was especially designed for shorter reads ($<$100 bp), we have considered D1 as input dataset but, for completeness, D2 is also included in the comparison. Figure 4.9 shows the alignment times using different number of mappers. In this case, each map process uses one core, so both terms, mappers and cores, are equivalent. Results show that SparkBWA clearly outperforms SEAL and pBWA for all the cases. As we have mentioned previously, SEAL times include the overhead caused by the preprocessing phase which takes on average about 1.9 and 2.9 minutes for D1 and D2 respectively. This overhead has a large impact on performance, especially for the smallest dataset.

The corresponding speedups obtained by the aligners for BWA-backtrack are displayed in Figure 4.10. As reference we have used the BWA sequential time. Results confirm the good behavior of SparkBWA with respect to SEAL and pBWA. For instance, SparkBWA reaches speedups up to 57$\times$ and 77$\times$ for D1 and D2 respectively. The maximum speedups achieved by SEAL are only about 31$\times$ and 42$\times$, while the corresponding values for pBWA are 46$\times$ and 59$\times$. In this way, SparkBWA is on average 1.9$\times$ and 1.4$\times$ faster than SEAL and pBWA respectively.

Finally, the BWA-MEM algorithm is evaluated considering the following tools: BWA, BigBWA, Halvade, and SparkBWA. Figure 4.11 shows the corresponding execution times for all the datasets varying the number of mappers (cores). BWA uses Pthreads in order to parallelize the alignment process, so it can only be executed on a single cluster node (64 cores). Both BigBWA and Halvade are based on Hadoop, and they require a preprocessing stage to prepare the input data for the alignment process. BigBWA requires, on average, 2.4, 5.8 and 23.6 minutes to preprocess each dataset, whereas Halvade spends 1.8, 6.6 and 22.7 minutes, respectively. Preprocessing is carried out sequentially for BigBWA, while Halvade is able to perform it in parallel. This overhead does not depend on the number of mappers used

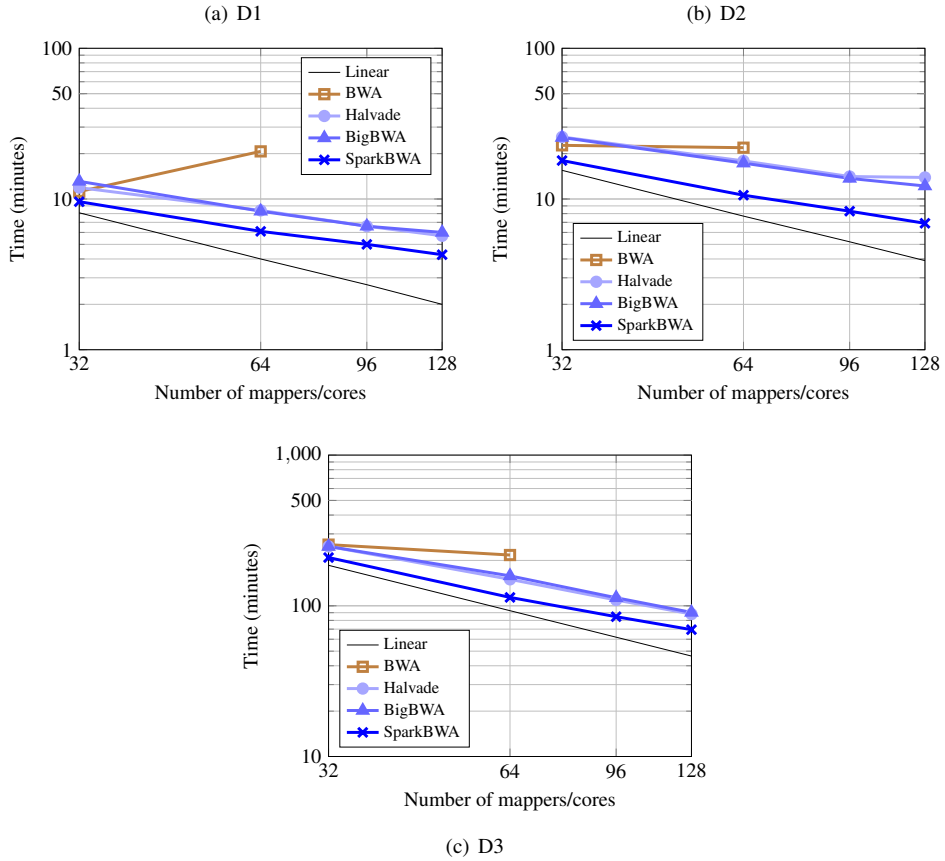


Figure 4.11: Execution times considering several BWA-based aligners running the BWA-MEM algorithm (axes are in log scale).

in the computations. For comparison fairness, the overhead of this phase is included in the corresponding execution times of both tools, since times for BWA and SparkBWA encompass the whole alignment process.

Performance results show that BWA is competitive with respect to Hadoop-based tools (BigBWA and Halvade) when 32 mappers are used, but its scalability is very poor. Using more threads in the computations do not compensate the overhead caused by their synchronization unless the dataset was big enough. BigBWA and Halvade show a better overall performance with respect to BWA. Both tools behave in a similar way, and differences in their performance

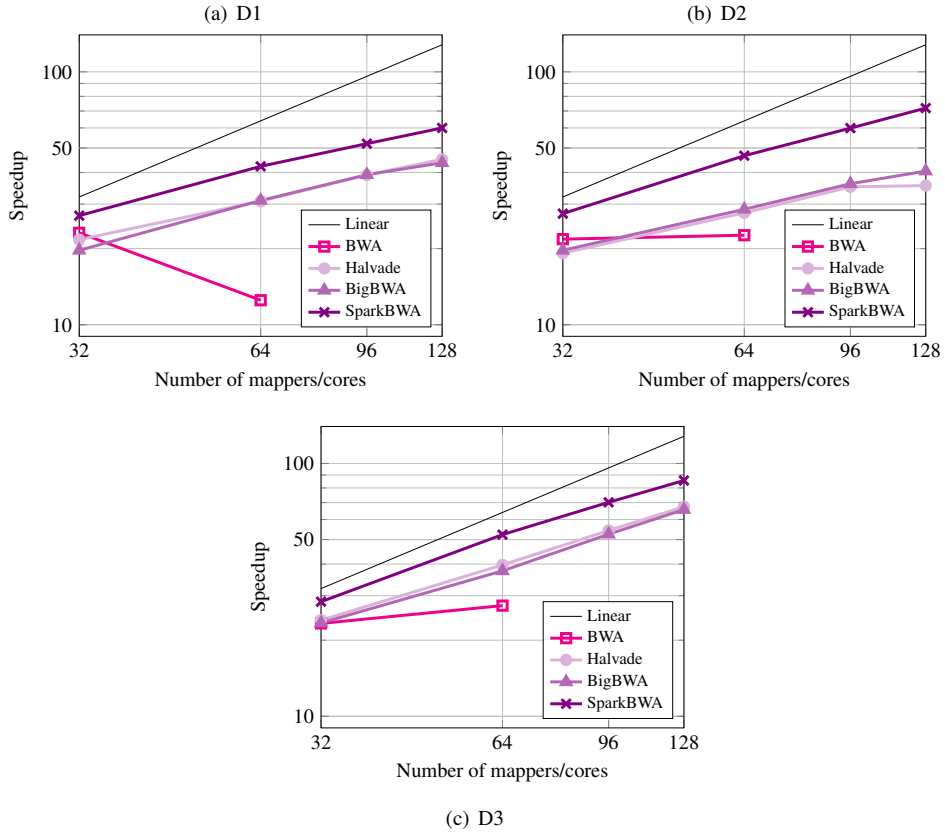


Figure 4.12: Speedup considering several BWA-based aligners running the BWA-MEM algorithm (axes are in log scale).

are small. Finally, SparkBWA outperforms all the considered tools. In order to illustrate the benefits of our proposal it is worth noting that, for example, SparkBWA is on average 1.5× faster than BigBWA and Halvade when using 128 mappers, and 2.5× with respect to BWA considering 64 mappers.

Performance results in terms of speedup with respect to the sequential execution of BWA are shown in Figure 4.12. The scalability problems of BWA are clearly revealed in the graphs. Hadoop-based tools show a better scalability but it is not enough to get closer to SparkBWA. The average speedup is respectively 50× and 49.2× for BigBWA and Halvade

using 128 workers. This value increases up to 72.5× for SparkBWA. Note that the scalability of SparkBWA is especially good when considering the biggest dataset (Figure 4.12(c)), reaching a maximum speedup of 85.6×. In other words, the parallel efficiency is 0.67.

In this way, SparkBWA has proven to be very consistent in all the scenarios considered, improving the results obtained by other state of the art BWA-based aligners. In addition, we must highlight that SparkBWA behaves better as the size of the dataset increases.

4.7. Conclusions

In this work we introduce SparkBWA, a new tool that exploits the capabilities of a Big Data technology as Apache Spark to boost the performance of the Burrows-Wheeler Aligner (BWA), which is a very popular software for mapping DNA sequence reads to a large reference genome. BWA consists of several algorithms especially tuned to deal with the alignment of short reads. SparkBWA was designed in such a way that no modifications to the original BWA source code are required. In this way, SparkBWA keeps the compatibility with any BWA software release, future or legacy.

The behavior of SparkBWA was evaluated in terms of performance, scalability and memory consumption. In addition, a thorough comparison between SparkBWA and several state of the art BWA-based aligners was performed. Those tools take advantage of different parallel approaches as Pthreads, MPI, and Hadoop to improve the performance of BWA. The evaluation shows that when considering the algorithm to align shorter reads (BWA-backtrack), SparkBWA is on average 1.9× and 1.4× faster than SEAL and pBWA. For longer reads and the BWA-MEM algorithm, the average speedup achieved by SparkBWA with respect to BigBWA and Halvade tools is 1.4×.

Finally, it is worth noting that most of the next-generation sequencing (NGS) professionals are not experts in Big Data or High Performance Computing. For this reason, in order to make SparkBWA more suitable for these professionals, an easy and flexible API is provided which will facilitate the adoption of the new tool by the community. This API allows to manage the sequence alignment process from the Apache Spark shell, hiding all the computational details to the users.

The source code of SparkBWA is publicly available at the GitHub repository (`https://github.com/citiususc/SparkBWA`).

Chapter 5

PASTASpark: multiple sequence alignment meets Big Data

Following is a reproduction of an article of which the author of this thesis is a main contributor. This is a verbatim reproduction, and the original can be found online under the following DOI: *10.1093/bioinformatics/btx354*, or with this information:

J. M. Abuín, T. F. Pena, and J. C. Pichel, "PASTASpark: multiple sequence alignment meets Big Data," *Bioinformatics*

5.1. Abstract

Motivation: One basic step in many bioinformatics analyses is the Multiple Sequence Alignment (MSA). One of the state of the art tools to perform MSA is PASTA (Practical Alignments using SATé and TrAnsitivity). PASTA supports multithreading but it is limited to process datasets on shared memory systems. In this work we introduce PASTASpark, a tool that uses the Big Data engine Apache Spark to boost the performance of the alignment phase of PASTA, which is the most expensive task in terms of time consumption.

Results: Speedups up to 10× with respect to single-threaded PASTA were observed, which allows to process an ultra-large dataset of 200,000 sequences within the 24-hr limit.

Availability: PASTASpark is an Open Source tool available at https://github.com/citiususc/pastaspark

Tag	Name	No. of sequences	Avg. sequence length	Size
D1	16S.B.ALL	27,643	1,371.9	184.1 MB
D2	50k RNASim	50,000	1,556	591.8 MB
D3	200k RNASim	200,000	1,556	3.4 GB

Table 5.1: Main characteristics of the input datasets.

5.2. Introduction

Multiple sequence alignment (MSA) is essential in order to predict the structure and function of proteins and RNAs, estimate phylogeny, and other common tasks in sequence analysis. PASTA [79] is a tool, based on SATé [80], which produces highly accurate alignments, improving the accuracy and scalability of other state-of-art methods, including SATé. PASTA is based on a workflow composed of several steps. During each phase, an external tool is called to perform different operations such as estimating an initial alignment and tree, computing MSAs on subsets of the original sequence set, or estimating the maximum likehood tree on a previously obtained MSA. Note that computing the MSAs is the most time consuming phase, implying in some cases over 70% of the total execution time.

PASTA is a multithreaded application that only supports shared memory computers. In this way, PASTA is limited to process small or medium size input datasets, because the memory and time requirements of large datasets exceed the computing power of any shared memory system. In this work we introduce PASTASpark, an extension to PASTA that allows to execute it on a distributed memory cluster making use of Apache Spark [13]. Apache Spark is a cluster computing framework that supports both in-memory and on-disk computations in a fault tolerant manner, using distributed memory abstractions known as Resilient Distributed Datasets (RDDs). PASTASpark reduces noticeably the execution time of PASTA, running the most costly part of the original code as a distributed Spark application. In this way, PASTASpark guarantees scalability and fault tolerance, and allows to obtain MSAs from very large datasets in reasonable time.

5.3. Approach

PASTA was written in Python and Apache Spark includes APIs for Java, Python, Scala and R. For this reason, authors use the Spark Python API (known as PySpark) to implement

PASTASpark. The design of PASTASpark minimizes the changes in the original PASTA code. In fact, the same software can be used to run the unmodified PASTA on a multicore machine or PASTASpark on a cluster: If Python is used, the original PASTA is launched, while if the job is submitted through Spark, PASTA is automatically executed in parallel using the Spark worker nodes.

The PASTA iterative workflow consists of four steps or phases. In the first phase (P1), a default starting tree is computed from the input sequence set S. Afterwards, using the tree and the centroid decomposition technique in SATé-II, S is divided into disjoint sets $S_1, \ldots, S_m$ and a spanning tree T^* on the subsets is obtained. In the second phase (P2), the MSAs on each S_i are obtained. By default, MAFFT [82] is used in this step, but other aligners could be employed. The resulting alignments are referred as *type 1 subalignments*. Now, in the third phase (P3), OPAL [123] is used to align the *type 1 subalignment* for every edge (v,w) in T^*, producing the so called *type 2 subalignment*, from which the final MSA is obtained through a sequence of pairwise mergers using transitivity. Finally, in the fourth phase (P4), if an additional iteration is desired, FastTree-2 [124] is executed to estimate a maximum likelihood tree on the MSA produced on the previous step, and the process is repeated using this tree as input.

As it was stated in [79], the most time consuming phase in PASTA is the computation of MSAs using MAFFT (P2). Due to this, PASTASpark focus on parallelizing this step. In the original PASTA, P2 is parallelized using Python multiprocessing. As MAFFT requires a file as input, the computed subsets S_i have to be stored in files from which each PASTA process calls the aligner using the Python class `Popen`. By default, PASTA creates so many processes as cores are available in the machine. This procedure has several important limitations: it only works on shared memory machines, it implies storing to disk a large amount of data, which could be a bottleneck, and, finally, it prevents MAFFT to run in parallel taking advantage of its OpenMP implementation.

To overcome these limitations, PASTASpark creates an in-memory RDD of key-value pairs, in case it detects that the Spark engine is running. In particular, the key is an object that includes information about the aligner and the required parameters to run it, while the value is an object containing a subset S_i. This RDD is automatically distributed by Spark to the worker nodes in the cluster, receiving each worker a slice of the RDD. Then, a map transformation is applied to the RDD. As a consequence, the input data is stored to each local disk of the worker nodes, and the aligner is invoked to process each local file using the `Popen`

		No. of workers (cores)				
		1	8	16	32	64
D1	PASTA	35.5	7.2 [69.5%]	–	–	–
	PASTASpark		7.4 [70.3%]	5.4 [59.3%]	4.5 [51.1%]	3.7 [40.6%]
D2	PASTA	100.5	18.7 [74.8%]	–	–	–
	PASTASpark		20.9 [77.5%]	14.4 [67.4%]	11.6 [59.5%]	9.5 [50.5%]

Table 5.2: Execution time (hours) for D1 and D2 using the CESGA cluster.

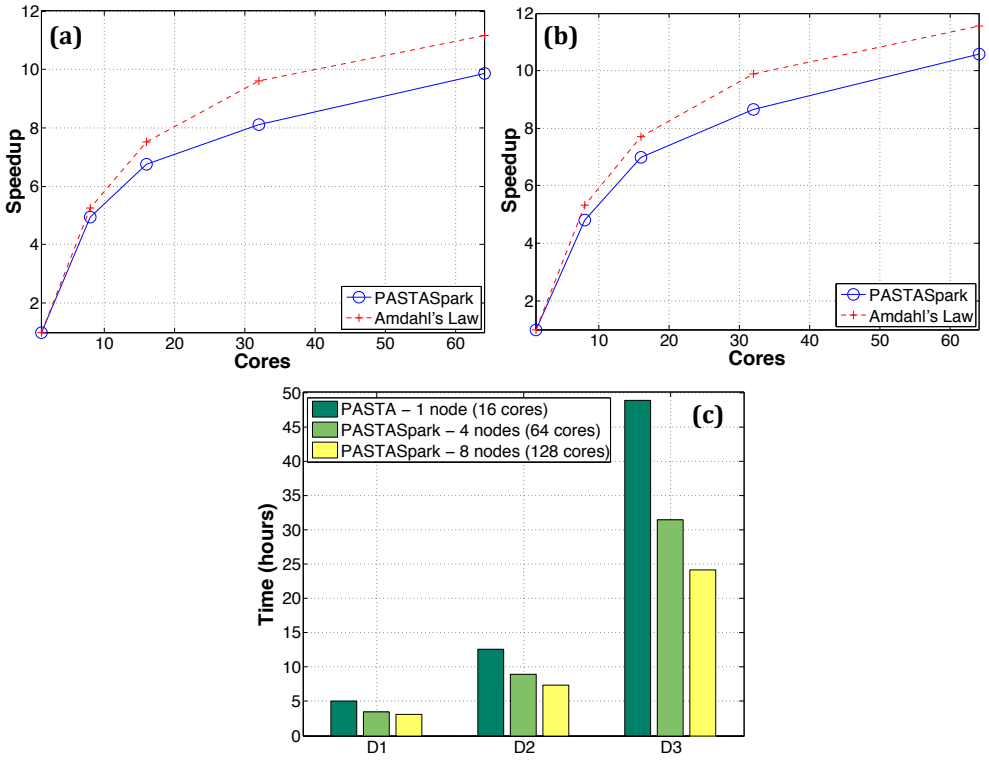


Figure 5.1: Speedup considering D1 (a) and D2 (b) datasets on the CESGA cluster, and execution times on the AWS cluster (c).

class. Notice that the amount of data stored to any of the local disks is divided by the number of workers. We must also highlight that the storing process is done in parallel, which reduces significantly the I/O cost. Besides, if the worker nodes are multicore machines, the aligner

software could run in parallel using several threads, which is also an important advantage of PASTASpark over the original PASTA application. The output of the map transformation is a new RDD that contains the *type 1 subalignments*. The resulting RDD is collected by the Spark Driver in order to continue the PASTA workflow. In this way, P2 is performed by the workers, while P1, P3 and P4 are executed by the Spark Driver. More details about PASTA and PASTASpark can be found in the Supplementary Material.

5.4. Results and discussion

Input datasets from the original PASTA publication are used to test PASTASpark performance. A summary of their main characteristics are shown in Table 5.1. Note that a starting tree is available for all the considered datasets, so its calculation in P1 is avoided. The experimental evaluation was carried out considering two different clusters, CESGA and AWS (see description in the Supplementary Material).

Table 5.2 shows the execution times of PASTA and PASTASpark when running on the CESGA cluster (PASTA can only run on a single 8 cores node). Important improvements were observed when using PASTASpark with different number of workers (cores). Note that the table displays between brackets the percentage of time spent in the alignment phase (P2), which was the one parallelized by PASTASpark. Those values are essential to understand the corresponding speedups achieved by PASTASpark with respect to original PASTA (see Figures 5.1(a) and 5.1(b)). In these figures we have used the Amdahl's law [125] to estimate the theoretical maximum speedup achievable by PASTASpark. This law states that if a fraction s of the work for a given problem can not be parallelized, with $0 < s \leq 1$, while the remaining portion, $1-s$, is p-fold parallel, then the maximum achievable speedup is $1/(s+(1-s)/p)$. In our particular case, P1, P3 and P4 phases in PASTA are multithreaded, so they could not be, in theory, considered sequential code. However, the execution time of P1 and P3 is really small with respect to P2 and P4, so without losing precision we can consider their execution time as a constant. On the other hand, the scalability of P4 (FastTree-2) is poor and it does not scale beyond 1.5–1.7$\times$ using 3 or more threads. Therefore, as a valid approximation for the current implementation of PASTA we consider P1, P3 and P4 as sequential processes. Red lines in Figures 5.1(a) and 5.1(b) show the Amdahl's law theoretical maximum speedup applied to D1 and D2 datasets. It can be observed that PASTASpark is close to the upper limit, obtaining speedups up to 10$\times$ when using 64 workers (cores).

Finally, Figure 5.1(c) displays the performance results of PASTASpark running on the AWS cluster using a different number of computing nodes. Each node in this system consists of 16 cores. It is worth to mention that PASTASpark is able to process an ultra-large dataset of 200,000 sequences (D3) within the 24 hour limit using only 8 AWS nodes.

5.5. Supplementary Material

5.5.1. Apache Spark

PASTASpark allows to execute PASTA on a distributed memory cluster using Apache Spark [13]. Spark is a cluster computing framework designed to support iterative jobs and interactive analytics, which includes automatic parallelization and task distribution as well as fault tolerance. Spark was originally developed at the University of California, Berkeley, becoming a Top-Level Apache Project in 2014. It uses a master/slave architecture with one central coordinator, named *Spark Driver*, and many distributed workers or *Spark Executors*, as Figure 5.2 illustrates. Spark applications run as independent sets of processes on a cluster coordinated by an object called *SparkContext*, which is a defined object in the Driver. The SparkContext can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos [25] or YARN [24]). Through the cluster manager, Spark acquires Executors on the cluster nodes, which are processes that run computations and store data for a Spark user program.

The Spark Driver is responsible for converting an user program into units of physical execution called *tasks*. A Spark program implicitly creates a logical directed acyclic graph (DAG) of operations, which the Driver converts into a physical execution plan consisting in a set of stages, where each stage is composed of multiple tasks. This plan is then optimized, for example, merging several transformations, and individual tasks are bundled up and prepared to be sent to the cluster. The Spark Driver tries to schedule each task in an appropriate location based on the data placement.

On the other hand, Spark Executors run the tasks that make up the application into a *Spark Container* (a Java Virtual Machine), returning results to the Driver. Besides, when tasks execute, their associated data can be cached in the Executors local memory. The Driver tracks the location of cached data and uses it to schedule future tasks that access that data.

Spark handle fault tolerance by introducing the concept of Resilient Distributed Datasets (RDDs) [39]. An RDD represents a read-only collection of objects partitioned across the

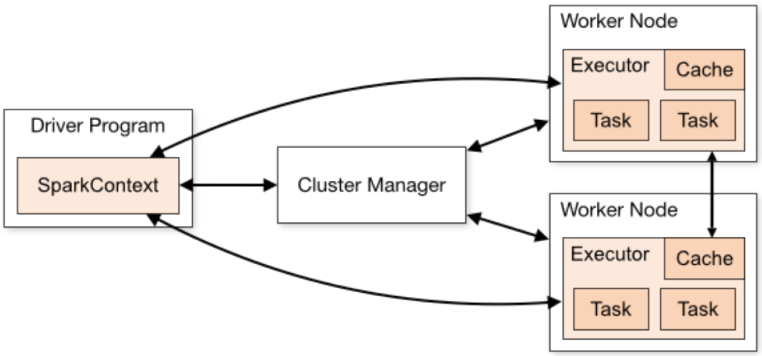


Figure 5.2: Apache Spark basic components.

cluster nodes that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. By using RDDs, programmers can perform iterative operations on their data without writing intermediary results to disk. In this way, Spark is well-suited, for example, to machine learning algorithms.

RDDs can be created from a collection of objects (e.g., a list or set) or by loading an external dataset. Note that Spark can process input data from HDFS, HBase [24], Cassandra [116], Hive [28], Parquet [117], and any Hadoop InputFormat. On created RDDs, Spark supports two types of parallel operations: transformations and actions. Transformations are operations on RDDs that return a new RDD, such as `map`, `filter`, `join`, `groupByKey`, etc. The resulting RDD will be stored in memory by default, but Spark also supports the option of writing RDDs to disk whenever necessary. On the other hand, actions are operations that kick off a computation, returning a result to the Driver program or writing it to storage. Examples of actions are `collect`, `count`, `take`, etc. Note that transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.

When running on a cluster, Spark allows two deploy modes namely *client* and *cluster* mode. In client mode, the Driver program runs on the same machine where it is itself being invoked, usually the user workstation. In cluster mode, the Driver will be shipped to execute on a worker node in the cluster, freeing the user computer to do other jobs.

Apache Spark provides Python, Scala, and R interactive shells, which let the user interact with data that is distributed on disk or in memory across many machines. Apart from running interactively, Spark can also be linked into applications in either Java, Scala, or Python. As the original PASTA is written in Python, we have decided to use the Spark Python API (also knows as PySpark) to implement PASTASpark.

Big Data and Bioinformatics

Due to the fast expansion of Big Data technologies, we can find several bioinformatics works that exploit the capabilities of different Big Data frameworks such as Apache Hadoop or Apache Spark on distributed memory clusters. However, to the best of our knowledge, none of those works deal with Multiple Sequence Alignment (MSA), being most of them focused on the short read alignment problem. Some relevant examples are the following:

- **SparkBWA** [78]: this tool exploits the capabilities of the Apache Spark engine to boost the performance of one of the most widely adopted aligner, the Burrows-Wheeler Aligner (BWA). The design of SparkBWA uses two independent software layers in such a way that no modifications to the original BWA source code are required, which assures its compatibility with any BWA version (future or legacy).

- **BigBWA** [76]: BigBWA is a tool that uses Apache Hadoop to boost the performance of the Burrows–Wheeler aligner (BWA). BigBWA is fault tolerant and it does not require any modification of the original BWA source code.

- **SEAL** [70]: SEAL is a scalable tool that also uses Apache Hadoop for short read pair mapping and duplicate removal. It computes mappings that are consistent with those produced by BWA and removes duplicates according to the same criteria employed by Picard's MarkDuplicates.

- **Halvade** [69]: Halvade is a framework that enables sequencing pipelines to be executed in parallel on a multi-node and/or multi-core Hadoop compute infrastructure in a highly efficient manner. As an example, a DNA sequencing analysis pipeline for variant calling has been implemented according to the GATK Best Practices recommendations, supporting both whole genome and whole exome sequencing.

Additionally, **MSAProbs-MPI** [84] is a distributed-memory parallel version of the multithreaded MSAProbs [83] tool that reduce runtimes by exploiting the compute capabilities

of distributed memory clusters by using the MPI [10] library. MSAProbs is a state-of-the-art protein multiple sequence alignment tool based on hidden Markov models. It can achieve high alignment accuracy at the expense of relatively long runtimes for large-scale input datasets. Note that unlike traditional parallel programming paradigms as MPI, code developing in Spark is largely simplified with characteristics as the automatic input splitting, task scheduling or fault tolerance mechanism.

5.5.2. PASTASpark in more detail

Next we describe the design and implementation of PASTASpark, but first, some background on the original PASTA application is provided.

PASTA Workflow

As it is described in [79], PASTA uses an iterative strategy. The first iteration begins with a starting tree, and subsequent iterations use as input the tree estimated in the previous one. In each step, the guide tree is used to divide the sequence set S into smaller subsets, and to build a spanning tree with these subsets as nodes. Then multiple sequence alignments (MSAs) for all the sequence subsets are independently estimated. After that, pairs of MSAs corresponding to subset that are adjacent in the spanning tree are aligned together. The resulting collection of MSAs overlap each other and are compatible where they overlap. These properties enable PASTA to merge these overlapping MSAs using transitivity and generate an MSA on the entire set of sequences. Finally, a maximum likelihood (ML) tree is estimated on the final alignment.

The starting tree can be provided by the user as an input parameter or it can be computed from an alignment A of a random subset X of 100 sequences from S. PASTA uses HMMER [126, 127] to compute an Hidden Markov Model on A, and to align all sequences in $S - X$ one by one to A. Then, an ML tree on this alignment is constructed using FastTree-2 [124].

Considering this starting tree as input, the PASTA workflow consist in an iterative method that we can summarize in four phases (see Figure 5.3):

Phase 1 (P1):

- The sequence set S is divided into disjoint sets $S_1, \ldots, S_m$, each with at most 200 sequences, using the current guided tree and the centroid decomposition technique in

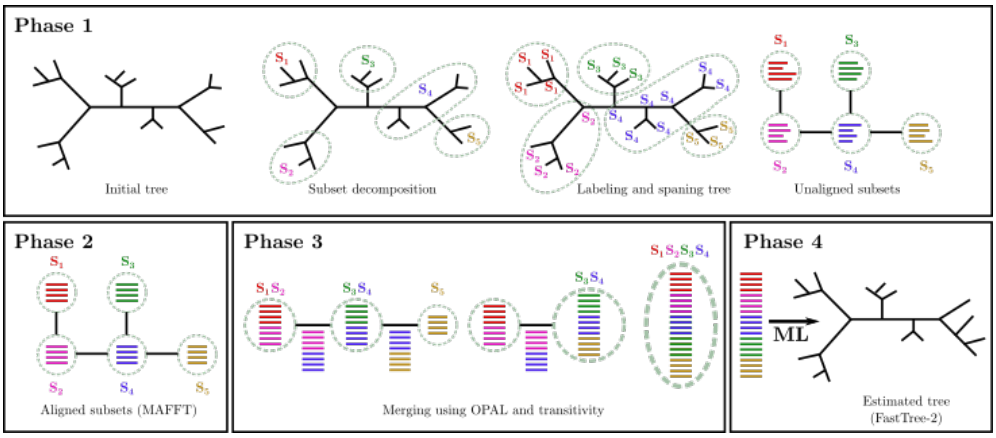


Figure 5.3: Main PASTA stages.

SATé-II [80]. In this technique, if the tree has at most 200 leaves, the set of sequences is returned; otherwise, an edge in the tree that splits the set of leaves into roughly equal sizes is found and removed from the tree. Then, the algorithm recurses on each subtree.

- A spanning tree T^* on the subsets is obtained. To do this, all leaves are labeled by their subset and, for every node v in the guide tree that is on a path between two leaves that both belong to S_i, it is labeled by S_i. If after this process there are unlabeled nodes, labels are propagated from nodes to unlabeled neighbors (breaking ties by using the closest neighbor according to branch lengths in the guide tree) until all nodes are labeled. Then, edges edges that have the same label at the endpoints are collapsed.

Phase 2 (P2):

- In this step, MSAs on each S_i are obtained using an existing MSA tool. Each such alignment is denoted as a *type 1 subalignment*. By default, MAFFT [82] with the L-INS-i settings is used. It is based on the iterative refinement method incorporating local pairwise alignment information.

Phase 3 (P3):

- Every node in T^* is labeled by an alignment subset for which we have a type 1 subalignment from previous step. For every edge (v, w) in T^*, OPAL [123] is employed to align the type 1 subalignments at v and w; this produces a new set of alignments, each

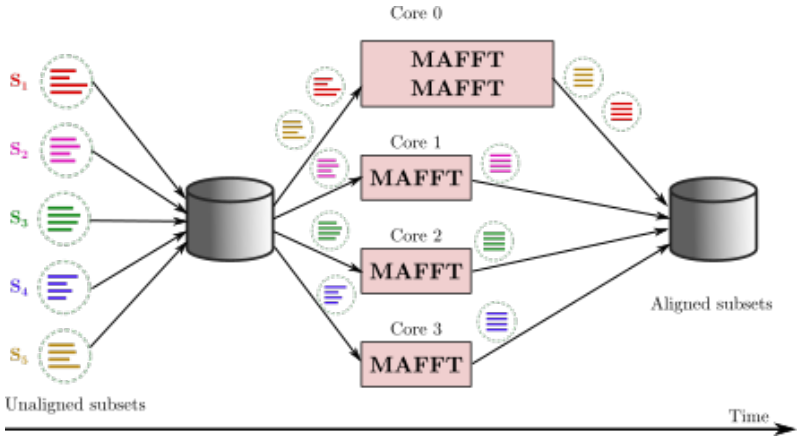


Figure 5.4: Phase 2 in PASTA.

containing at most $2k$ sequences, which are called *type 2 subalignments*. The merger technique used to compute type 2 subalignments is required not to change the alignments on the type 1 subalignments; therefore, type 2 subalignments induce the type 1 subalignments computed in the phase 2 and are all compatible with each other.

- The final alignment is computed through a sequence of pairwise mergers using transitivity, as described in [79]. Using the concept of *transitivity merger*, the spanning tree is used to merge all the type 2 subalignments through a sequence of pairwise transitivity mergers into a multiple sequence alignment on the entire set of sequences. The final transitivity merger produces an alignment that includes all the sequences.

Phase 4 (P4):

- If an additional iteration (or a tree on the alignment) is desired, FastTree-2 is used to estimate a maximum likelihood tree on the MSA produced in the previous phase and the process is repeated.

As it was stated in [79], P1 and P3 require a small amount of time, so their parallelization/optimization is not necessary. The most expensive phase in terms of computational time is P2 (subsets alignment), which may imply more than 70% of the total execution time. Finally, P4 (tree estimation) requires more time than P1 and P3 but it is much faster than P2.

P2 in PASTA is parallelized using a multithreaded approach. By default, PASTA set the number of threads to be used equal to the number of available cores in the shared memory system, although the user can also specify a particular value. Figure 5.4 shows an example of how P2 is implemented in PASTA considering five subsequences and a quad-core computer. First, the subsequences are stored in the hard disk because the input to the aligner (MAFFT in this case) can only be a regular file. Afterwards, each Python thread forks a new child process, which execute MAFFT using Python's class `subprocess.Popen`. These child processes run in parallel in the four cores, and their outputs are again stored to disk to be used in the next phase (P3). Note that in the example of Figure 5.4, since there are five subsequences and only four cores, one of the cores executes sequentially two MAFFT subprocesses.

The PASTA parallelization strategy has three main drawbacks:

1. Both the Python reference interpreter (CPython) and the alternative interpreter that offers the fastest single-threaded performance for pure Python code (PyPy), use a Global Interpreter Lock (GIL) [128] to avoid various problems that arise when using threading models that implicitly allowing concurrent access to objects from multiple threads of execution. This makes Python unsuitable for using shared memory threading to exploit multiple cores on a single machine. To overcome this problem, the adopted solution in PASTA was to lunch a child process in each thread using the Python multiprocessing library. These processes can run in parallel but with the extra overhead of creating them.

2. The second drawback is that, with the adopted approach, each subprocess runs on a single core and call the external alignment program (MAFFT by default) with one of the input subsequences. MAFFT supports multithreading with OpenMP, but, as each subsequence is aligned in a single core, MAFFT parallelism cannot be harnessed.

3. The third drawback and the most important one is that PASTA can only be executed on shared memory computers, which limits its scalability to a few number of cores. In this way, PASTA is limited to process small input datasets because the memory and time requirements of large datasets exceed the computing power of any shared memory system.

PASTASpark

The objective of PASTASpark is to reduce the execution time of PASTA using Apache Spark as engine to run it on a distributed memory cluster. Current version of PASTASpark

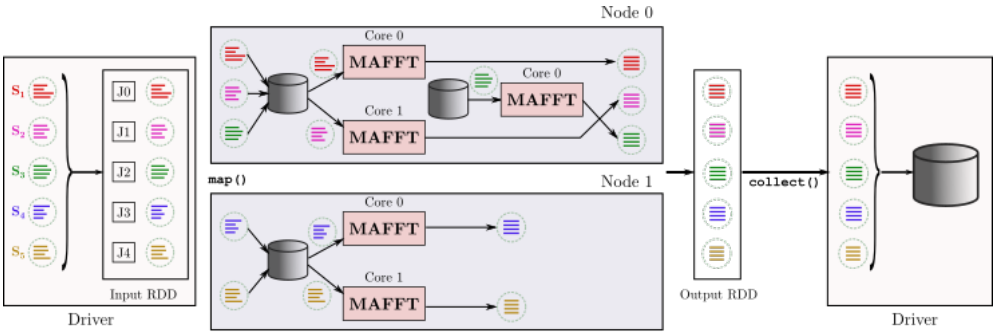


Figure 5.5: Phase 2 in PASTASpark.

focused only in the subset alignment step (P2) of the PASTA workflow. So, other steps are executed by the Spark Driver program.

Figure 5.5 shows how P2 is modified in PASTASpark. This example considers that we have again five subsequences and Spark is running on a distributed memory cluster with two dual core nodes. If the code is launched using `spark_submit`, the output of the first phase is not spilled to disk but a pair RDD is created. A pair RDD is an RDD where the elements are (*key*, *value*) pairs. In this case, the key is a job identifier and the value is one of the subsequences created in P1 and which resides in the Driver memory. Afterwards, a `map()` transformation is applied to the RDD in such a way that MAFFT is executed concurrently on each of the subsequences. We must highlight that each piece of the RDD is stored in the local disk of a worker node (not in the Driver disk), and the writing is performed in parallel reducing the I/O cost with respect to PASTA.

The output of the `map` transformation is a new RDD with the aligned subsequences (stored in the local memory of the worker nodes). As transformations on RDDs are lazily evaluated, Spark will not start to execute the `map` until it detects an action. In this case, a `collect()` action is used to retrieve the output RDD as a list, which is finally stored in the local disk of the Driver. From this point, the rest of the PASTA phases are executed in the Driver.

Tag	Name	No. of sequences	Avg. sequence length	Size
D1	16S.B.ALL	27,643	1,371.9	184.1 MB
D2	50k RNASim	50,000	1,556	591.8 MB
D3	200k RNASim	200,000	1,556	3.4 GB

Table 5.3: Datasets used for the experimental evaluation of PASTASpark.

5.5.3. Experimental setup

Input Datasets

In the original PASTA paper [79] several input datasets were used. In particular, datasets are from the million-sequence RNASim [129] with subsamples of 10k, 20k, 50k, 100k and 200k sequences, and biological datasets from the comparative ribosomal website (CRW) [130]. In order to test PASTASpark datasets from both groups have been chosen (see Table 5.3 for details).

According to the original PASTA paper and regarding the RNASim datasets, PASTA was able to complete two iterations with the 100k subsample and only one iteration with the 200k subsample. However, it completes the three iterations that PASTA runs by default with the 50k sample. For this reason we have chosen as illustrative examples the 50K and 200k datasets (D2 and D3 respectively). D1 was selected since it is the biggest dataset among the CRW examples.

Computational Platforms

We have used two different clusters in the performance tests. The first one is a Big Data cluster installed at CESGA[1] (Galicia Supercomputing Center in Spain). This cluster has 12 computing nodes with 54.5 GB of RAM memory each one, and 19 cores available for Spark containers. However, in our case the maximum number of cores that can be used per container is 8 due to some restrictions set by the system administrators. CPUs are Intel Xeon CPU E5-2620-v3 at 2.40GHz. Hadoop, Spark and Java versions are 2.7.1 (HDP), 1.6.1, and 1.8, respectively.

The second platform is a 9-node AWS (Amazon Web Services) EC2 cluster[2] with 16 cores (Intel Xeon E5-2670 at 2.5GHz CPUs) and 122 GB of RAM memory per node. In particular,

[1] www.cesga.es

[2] aws.amazon.com/ec2/

each computing node corresponds to a r3.4xlarge EC2 instance. In this case, Hadoop, Spark and Java versions are 2.7.2, 1.6.2, and 1.8 respectively.

It should be taken into account that PASTASpark runs P1, P3 and P4 in the Spark Driver, while P2 is performed by the Spark Executors. In this way, depending on the input dataset some hardware parameters should be set: the number of cores and RAM memory used by the Driver, the memory assigned to the Executors, etc. For example, the Driver considering `D2` uses 20 GB, while for `D1` 15 GB is an adequate value. Regarding the Executors, 5 GB are enough in both cases.

5.5.4. Future work

The first topic to be addressed in the future work is the problem related to the parallelization efficiency of FastTree-2 tool (P4). As the number of cores used in the computation increases, P4 becomes the major bottleneck in the performance of PASTASpark. In this way, it is necessary to completely redesign the parallelization strategy of FastTree-2 or Fast-Tree-2 should be replaced by a different tool with a better scalability. The goal is also to integrate P4 in the Spark framework. In this way, we believe that the scalability of PASTASpark will improve noticeably.

CHAPTER 6

CONCLUSIONS

In the recent years Big Data tools and ecosystems have become the standard when analyzing or processing huge datasets. We find the cause in the benefits of these technologies: fault tolerance, use of high level programming languages, or their similarity with High Performance Computing (HPC) regarding its parallel philosophy. Many experts in the HPC area agree that the Big Data and the HCP ecosystems should converge, or at least, share some approaches in order to enter into the Exascale era. In this way, HPC should incorporate some features from Big Data technologies such as the fault tolerance or the fast data distribution.

In this thesis, we use Big Data technologies to deal with some scientific problems that are computationally intensive regarding execution time (typical in HPC problems) and, at the same time, have a large input data size (typical in Big Data), with the objective of improving the execution time, scalability and efficiency. Conclusions derived from this work, and presented in this chapter, can clarify where the barrier between these two paradigms stands, or even prove whether this barrier exists at all. In this way, we want to contribute to solve the key question that in recent times has arisen among the HPC community could be solved: should Big Data be considered part of the High Performance Computing field?

Next, the main conclusions derived from this work are summarized:

- The field of NLP needs scalable and efficient tools in order to process the huge amount of information available in the Big Data era. However, the state of the art tools are usually written in languages that are not suitable neither for HPC nor Big Data technologies. This is the case, for example, of the existent modules in the Linguakit repository, written in Perl language. Although Hadoop provides the Hadoop Streaming tool

to deal with these cases, it has shown a poor performance. To overcome this issue, we have developed Perldoop (Chapter 2), which is a tool that automatically translates Perl scripts prepared to be executed using Hadoop Streaming into Hadoop-ready Java codes. However, the objective was not to develop a powerful tool that allows to automatically translate any existent Perl code to Java, but a simple and easy-to-use tool that takes as input Perl codes written for Hadoop Streaming, follows a reduced number of additional programming rules, and produces Hadoop-ready Java codes. By using this tool, NLP programs can benefit from the scalability, performance and fault tolerance properties of Big Data technologies. To facilitate this job, the tool uses a system based on tagging the source code and templates. Results show how, by using the MapReduce-ready Java codes generated by Perldoop, the NERC modules from Linguakit are executed $8\times$ faster than using the Hadoop Streaming tool with the same number of CPUs. For instance, the original NERC modules require about 19 days to process the whole Wikipedia in Spanish language. This result is improved by the Hadoop Streaming tool to less than 16 hours when using 64 cores. However, considering the Perldoop generated codes, the time is noticeably reduced to less than 2 hours using the same number of cores.

- In the last years, the available biological data in a digital format has experimented a big and fast growth. This process has been possible thanks to the next-generation sequencing (NGS) technologies. These technologies have facilitated the extraction of huge quantities of DNA sequences, which will further promote the future growth of biological databases. However, the process of giving meaning to all this information is overcoming the computing capacity of a CPU. Due to this fact, bioinformatics tools need to be efficient and scalable; that is, they need to deal with an ever growing amount of data. One of the most important challenges in Bioinformatics is the sequence alignment process. In this thesis, we deal with this problem introducing the BigBWA tool (Chapter 3). BigBWA uses Hadoop as a Big Data technology, while internally it uses the C functions from the state of the art software BWA to perform the alignment phase. In this way, two independent software layers were created in BigBWA. The first one corresponds to the BWA software package, while the other is, strictly speaking, our tool. This design implies that no modification of the BWA source code is needed, which assures the compatibility of BigBWA with any BWA version. Also, BigBWA is fault tolerant. It is worth noting that considering the 6 billion (6×10^9) reads that a Illumina HiSeqX^{TM} Ten is able to generate, BigBWA is capable or performing the alignment in

just 5 hours considering a medium size cluster. That is, 192× faster than single-thread BWA.

- The Big Data world evolves very quickly as Apache Spark illustrates. Since its birth, destined to overcome the Apache Hadoop limitations, it has experimented an enormous growth regarding functionalities and improvements. Taking advantage of this technology, we have developed SparkBWA (Chapter 4). SparkBWA follows the philosophy of BigBWA in terms of software design. SparkBWA fulfills three requirements. First, SparkBWA outperforms BWA and other BWA-based aligners both in terms of performance and scalability. Second, it keeps the compatibility with future and legacy versions of BWA. Since BWA is constantly evolving to include new functionalities and algorithms, it is important for SparkBWA to be agnostic regarding the BWA version. This is an important difference with respect to other existing tools based on BWA, which require modifications of the BWA source code. Finally, NGS professionals demand solutions to perform sequence alignments efficiently, in such a way that the implementation details are completely hidden to them. For this reason SparkBWA provides a simple and flexible API to handle all the aspects related to the alignment process, which allows bioinformaticians to focus only on the scientific problem to deal with. In terms of performance, we must highlight that SparkBWA is almost twice faster than other state of the art tools.

- Multiple Sequence Alignment (MSA) is an extension of the pairwise alignment to incorporate more than two sequences at a time. Multiple sequence alignments are computationally difficult to produce and most formulations of the problem lead to NP-complete combinatorial optimization problems. PASTA is a multithreaded application which produces highly accurate alignments, improving the accuracy and scalability of other state-of-art methods. PASTA is limited to process small or medium size input datasets, because the memory and time requirements of large datasets exceed the computing power of any shared memory system. In this thesis, we introduce PASTASpark (Chapter 5), an extension to PASTA that allows to execute it on a distributed memory cluster making use of Apache Spark. The design of PASTASpark minimizes the changes in the original PASTA code. In fact, the same software can be used to run the unmodified PASTA on a multicore machine or PASTASpark on a cluster. We are able to process a dataset composed of 200.000 sequences in less than 24 hours. Note that the original PASTA tool can not complete this process because of memory restrictions.

- We consider that a two level programming model can be a solution for mixing the Big Data and HPC worlds. A high level programming interface should be used when accessing the data for reading or writing (for example, by using HDFS) with languages such as Java, Scala or Python, more suited for these kind of tasks. This should be complemented with a low level programming interface when dealing with performance, memory consumption restrictions, or high performance libraries. Here, the programming languages should be the classic HPC languages, such as C, C++ or Fortran. In this way, advantages from the two worlds can be obtained, with a minimum performance penalty, as we have demonstrated in Chapter 3 and 4. We have demonstrated that the efficient communication of these two layers can be performed by using the Java Native Interface (JNI), or calling native methods from Python.

- When applications are not written in languages suited for Big Data nor HPC technologies, the best option is porting the codes to another language. However, this is a tedious and hard job. In this way, source to source compilers can be a great help, as the case shown in Chapter 2.

- Memory consumption could be a problem in Big Data applications. This is mainly caused by the way containers are implemented in YARN. A container is basically a process running as a Java Virtual Machine (JVM), with the corresponding memory overhead and memory management issues related to the Java Garbage Collector. A solution for this can be to use JNI as well, since native methods can reserve and free memory that the Java Garbage Collector is not aware of. This is a good approach, but at the same time, it requires that the programmer reserves and frees memory according to the program requirements.

6.1. Future work

We present here a list of future tasks that can be carried out in order to continue the work started in this thesis:

- Current Big Data schedulers do not allow to incorporate resources such as GPUs or accelerators such as Intel Xeon Phi, which are typical components of current supercomputing nodes. Add them as resources in YARN or Mesos should be a great step.

- Metagenomic sequencing studies are becoming increasingly popular, including the sequencing of human microbiomes and diverse environments. A fundamental computational problem in this context is read classification. For example, the assignment of each read to a taxonomic label. Due to the large number of reads produced by modern high-throughput sequencing technologies and the rapidly increasing number of available reference genomes, current software tools suffer from either long runtimes, large memory requirements and/or low accuracy. Because of this, Big Data technologies seem suitable for this problem. A possibility for future works is to enter into the metagenomics scientific field.

- Development of a new Perldoop version. This new version will incorporate the possibility of translate codes into other kind of Big Data technologies, and not only Apache Hadoop. Among the possible candidates we can highlight Apache Spark and Apache Storm.

- PASTA has another bottleneck that could be improved. This bottleneck is the starting tree construction, which is performed by the FastTree-2 tool and only scales up until 4 cores. If this construction phase could be improved, it would cause a great impact in the field of Multiple Sequence Alignment. In order to do so, the construction algorithm should be completely redesigned.

- BigBWA, SparkBWA and PASTASpark perform some writing/reading to/from disk, with the consequent lose of performance. These operations could be avoided by passing the input sequence data directly to the BWA and PASTA functions from the Big Data side of the application. In order to do that, a C library to communicate the Big Data side of the application with its C codes counterparts needs to be created.

Bibliography

[1] IBM, "Big Data at the Speed of Business," http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html, [Online; accessed July, 2014].

[2] Cern, "CERN Web Site," https://home.cern, [Online; accessed May, 2017].

[3] "Square Kilometre Array Home Page," http://skatelescope.org/, [Online; accessed June, 2017].

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[5] ——, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] Hadoop, "Apache Hadoop," http://hadoop.apache.org, [Online; accessed May, 2017].

[7] TOP500, "TOP500 List," https://www.top500.org/, [Online; accessed May, 2017].

[8] Cesga, "Galicia Supercomputing Centre Web Site," http://www.cesga.es/, [Online; accessed May, 2017].

[9] E. exchange, "Processing power compared," http://pages.experts-exchange.com/processing-power-compared/, [Online; accessed May, 2017].

[10] D. W. Walker and J. J. Dongarra, "MPI: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[11] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[12] "OpenACC web page," https://www.openacc.org/, [Online; accessed Apr, 2017].

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010, pp. 10–10.

[14] H. Asaadi, D. Khaldi, and B. M. Chapman, "A Comparative Survey of the HPC and Big Data Paradigms: Analysis and Experiments," in *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, 2016, pp. 423–432.

[15] D. A. Reed and J. Dongarra, "Exascale Computing and Big Data," *Commun. ACM*, vol. 58, no. 7, pp. 56–68, Jun. 2015.

[16] Lustre, "Lustre File System," http://lustre.org, [Online; accessed May, 2017].

[17] G. Staples, "TORQUE Resource Manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[18] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[19] "Nvidia CUDA Home Page," http://www.nvidia.com/object/cuda_home_new.html, [Online; accessed May, 2017].

[20] "OpenCL Home Page," https://www.khronos.org/opencl/, [Online; accessed May, 2017].

[21] "LAPACK web page," http://www.netlib.org/lapack/, [Online; accessed Apr, 2017].

[22] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of the 4th Annual Symposium on Cloud Computing (SOCC)*, 2013, pp. 5:1–5:16.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 295–308.

[26] T. White, *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly Media, Inc., 2012.

[27] "Apache Pig Home Page," https://pig.apache.org/, [Online; accessed June, 2017].

[28] "Apache Hive home page," http://hive.apache.org/, [Online; accessed December, 2016].

[29] HBase, "HBase," https://hbase.apache.org/, [Online; accessed May, 2017].

[30] I. F. Haddad, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, vol. 2000, no. 80es, Nov. 2000.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43.

[32] M. K. McKusick and S. Quinlan, "GFS: Evolution on Fast-forward," *Queue*, vol. 7, no. 7, pp. 10:10–10:20, Aug. 2009.

[33] T. White, *Hadoop: The Definitive Guide*, 4th ed. O'Reilly Media, Inc., 2015.

[34] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting Scientific Computing Problems to Clouds Using MapReduce."

[35] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for Hadoop," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 4.

[36] "Cascading home page," http://www.cascading.org/, [Online; accessed February, 2016].

[37] M. Ding, L. Zheng, Y. Lu, L. Li, S. Guo, and M. Guo, "More convenient more overhead: the performance evaluation of Hadoop streaming," in *ACM Symp. on Research in Applied Computation*, 2011, pp. 307–313.

[38] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analytics*, 1st ed. O'Reilly Media, Inc., 2015.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.

[40] R. Agerri, X. Artola, Z. Beloki, G. Rigau, and A. Soroa, "Big data for natural language processing: a streaming approach," *Knowledge-Based Systems*, vol. 79, pp. 36–42, 2015.

[41] P. Gamallo and M. García, "A Resource-Based Method for Named Entity Extraction and Classification," *LNCS series*, vol. 7026, pp. 610–623, 2011.

[42] T. Hasegawa, S. Sekine, and R. Grishman, "Discovering relations among named entities from large corpora," in *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2004, p. 415.

[43] Z. Kozareva, J. Silva, P. Gamallo, and G. Lopes, "Cluster analysis of named entities," in *Intelligent Information Processing and Web Mining*. Springer, 2004, pp. 429–433.

[44] J. Nothman, N. Ringland, W. Radford, T. Murphy, and J. R. Curran, "Learning multilingual named entity recognition from Wikipedia," *Artificial Intelligence*, vol. 194, pp. 151–175, 2013.

[45] D. Lin and X. Wu, "Phrase clustering for discriminative learning," in *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*. Association for Computational Linguistics, 2009, pp. 1030–1038.

[46] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2005, pp. 363–370.

[47] M. Garcia and P. Gamallo, "An Entity-Centric Coreference Resolution System for Person Entities with Rich Linguistic Information," in *COLING*, 2014, pp. 741–752.

[48] ——, "Yet another suite of multilingual NLP tools," in *International Symposium on Languages, Applications and Technologies*. Springer, 2015, pp. 65–75.

[49] J. M. Abuín, J. C. Pichel, T. F. Pena, P. Gamallo, and M. Garcia, "Perldoop: Efficient execution of Perl scripts on Hadoop clusters," in *IEEE International Conference on Big Data*, 2014, pp. 766–771.

[50] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank," *Nucleic Acids Research*, vol. 41, no. D1, p. D36, 2013.

[51] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L. L. Yeh, "UniProt: the Universal Protein knowledgebase," *Nucleic Acids Research*, vol. 32, no. S1, pp. 115–119, 2004.

[52] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome research*, vol. 20, no. 2, pp. 265–272, 2010.

[53] J. Shendure, R. D. Mitra, C. Varma, and G. M. Church, "Advanced sequencing technologies: methods and goals," *Nature Reviews Genetics*, vol. 5, no. 5, pp. 335–344, 2004.

[54] B. Schmidt, *Bioinformatics: High Performance Parallel Computer Architectures*, 1st ed. CRC Press, 2011.

[55] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, pp. 1297–1303, Sep 2010.

[56] B. institute, "GATK Best Practices," https://software.broadinstitute.org/gatk/best-practices/, [Online; accessed May, 2017].

[57] D. W. Mount, *Bioinformatics: sequence and genome analysis*. CSHL press, 2004, ch. Phylogenetic Prediction.

[58] R. C. Deonier, S. Tavaré, and M. Waterman, *Computational genome analysis: an introduction*. Springer Science & Business Media, 2005.

[59] B. Haubold and T. Wiehe, *Introduction to computational biology: an evolutionary approach*. Springer Science & Business Media, 2006.

[60] N. C. Jones and P. Pevzner, *An introduction to bioinformatics algorithms*. MIT press, 2004.

[61] H. Li and R. Durbin, "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[62] ——, "Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[63] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv:1303.3997v2*, 2013.

[64] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.

[65] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, p. 713, 2008.

[66] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, p. 1966, 2009.

[67] C.-M. Liu, T. K. F. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. W. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[68] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. Lam, "SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner," *PLoS ONE*, vol. 8, no. 5, 2013.

[69] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier, "Halvade: Scalable Sequence Analysis with MapReduce," *Bioinformatics*, vol. 31, no. 15, pp. 2482–2488, 2015.

[70] L. Pireddu, S. Leo, and G. Zanetti, "SEAL: a distributed short read mapping and duplicate removal tool," *Bioinformatics*, vol. 27, no. 15, pp. 2159–2160, 2011.

[71] S. Leo and G. Zanetti, "Pydoop: a Python MapReduce and HDFS API for Hadoop," in *Proc. of 19th Symposyum on HPDC*, 2010, pp. 819–825.

[72] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 2012.

[73] Y. Liu and B. Schmidt, "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing," *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2014.

[74] Y. Liu, B. Popp, and B. Schmidt, "CUSHAW3: sensitive and accurate base-space and color-space short-read alignment with hybrid seeding," *PloS one*, vol. 9, no. 1, p. e86869, 2014.

[75] D. Peters, X. Luo, K. Qiu, and P. Liang, "Speeding Up Large-Scale Next Generation Sequencing Data Analysis with pBWA," *Journal of Applied Bioinformatics & Computational Biology*, vol. 1, no. 1, pp. 1–6, 2012.

[76] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "BigBWA: Approaching the Burrows–Wheeler Aligner to Big Data Technologies," *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015.

[77] S. Liang, *Java Native Interface: Programmer's Guide and Reference*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[78] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "SparkBWA: speeding up the alignment of high-throughput DNA sequencing data," *PloS one*, vol. 11, no. 5, p. e0155461, 2016.

[79] Mirarab, S. *et al.*, "PASTA: ultra-large multiple sequence alignment for nucleotide and amino-acid sequences," *Journal of Computational Biology*, vol. 22, no. 5, pp. 377–386, 2015.

[80] Liu, K. *et al.*, "SATé-II: very fast and accurate simultaneous estimation of multiple sequence alignments and phylogenetic trees," *Systematic Biology*, vol. 61, no. 1, pp. 90–106, 2012.

[81] J. M. Abuín, T. F. Pena, and J. C. Pichel, "PASTASpark: multiple sequence alignment meets Big Data," *Bioinformatics*.

[82] Katoh, K. *et al.*, "MAFFT: improvement in accuracy of multiple sequence alignment," *Nucleic acids research*, vol. 33, no. 2, pp. 511–518, 2005.

[83] Y. Liu, B. Schmidt, and D. L. Maskell, "MSAProbs: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities," *Bioinformatics*, vol. 26, no. 16, pp. 1958–1964, 2010.

[84] J. González-Domínguez, Y. Liu, J. Touriño, and B. Schmidt, "MSAProbs-MPI: parallel multiple sequence aligner for distributed-memory systems," *Bioinformatics*, vol. 32, no. 24, pp. 3826–3828, 2016.

[85] A. Gudyś and S. Deorowicz, "QuickProbs—A Fast Multiple Sequence Alignment Algorithm Designed for Graphics Processors," *PLOS ONE*, vol. 9, no. 2, pp. 1–18, 02 2014.

[86] P. Gamallo, J. C. Pichel, M. Garcia, J. M. Abuín, and T. F. Pena, "Análisis morfosintáctico y clasificación de entidades nombradas en un entorno Big Data," *Procesamiento del Lenguaje Natural*, vol. 53, pp. 17–24, 2014.

[87] J. Kegler, "Perl And Undecidability: The Halting Problem," *The Perl Review*, vol. 4, pp. 21–25, 2008.

[88] ——, "Perl And Undecidability: Rice's Theorem," *The Perl Review*, vol. 4, pp. 23–29, 2008.

[89] L. Wall, B. Jepson, N. Patwardhan, E. Siever, and D. Futato, *PERL Resource Kit UNIX Edition: 4 Volume Set with CD-ROM*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1997.

[90] "Perl programming documentation," http://perldoc.perl.org/, [Online; accessed July, 2014].

[91] Oracle, "Java Platform, Standard Edition 7 API Specification," http://docs.oracle.com/javase/7/docs/api/java, [Online; accessed July, 2014].

[92] JRegex, Regular Expressions for Java, http://jregex.sourceforge.net/, [Online; accessed July, 2014].

[93] P. Gamallo and M. García, "Using Morphosyntactic Post-processing to Improve PoS-tagging Accuracy," in *9th Int. Conf. on Computational Processing of Portuguese Language (PROPOR)*, 2010.

[94] M. Banko and R. Moore, "Part of Speech Tagging in Context," in *Proc. of the 20th Int. Conf. on Computational Linguistics*, 2004.

[95] L. Padró and E. Stanilovsky, "Freeling 3.0: Towards wider multilinguality," in *Proc. of the LREC*, 2012.

[96] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proc. of the ACM SIGMOD Int. Conf. on Management of data*, 2008, pp. 1247–1250.

[97] J. Lehman et al., "DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia," *Semantic Web Journal*, 2014.

[98] L. Lai et al., "ShmStreaming: A Shared Memory Approach for Improving Hadoop Streaming Performance," *Int. Conf. on Advanced Information Networking and Applications*, pp. 137–144, 2013.

[99] E. Dede et al., "MARISSA: MApReduce Implementation for Streaming Science Applications," in *eScience*, 2012, pp. 1–8.

[100] C. Dyer, A. Cordova, A. Mont, and J. Lin, "Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce," in *Proc. of the Workshop on Statistical Machine Translation*, 2008, pp. 199–207.

[101] R. Ahmad, P. Kumar, B. Rambabu, P. Sajja, M. K. Sinha, and R. Sangal, "Enhancing Throughput of a Machine Translation System using MapReduce Framework: An Engineering Approach," in *9th Int. Conf. on Natural Language Processing*, 2011.

[102] J. Lin, "Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce," in *Proc. of the EMNLP*, 2008, pp. 419–428.

[103] P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu, and V. Vyas, "Web-scale distributional similarity and entity set expansion," in *Proc. of the EMNLP*, 2009, pp. 938–947.

[104] D. Metzler and E. Hovy, "Mavuno: a scalable and effective Hadoop-based paraphrase acquisition system," in *Proc. of the 3rd Workshop on Large Scale Data Mining: Theory and Applications*, 2011, p. 3.

[105] "Integrating NLTK with the Hadoop MapReduce Framework."

[106] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proc. of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.

[107] S. Bird, "NLTK: the natural language toolkit," in *Proc. of the COLING/ACL on Interactive presentation sessions*, 2006, pp. 69–72.

[108] G. Attardi, S. D. Rossi, and M. Simi, "The Tanl Pipeline," in *Proc. of the 7th Int. Conf. on Language Resources and Evaluation*, may 2010.

[109] Altshuler, D. et al., "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, pp. 1061–1073, 2010.

[110] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic Acids Research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[111] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[112] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "BarraCUDA - a fast short read sequence aligner using graphics processing units," *BMC Research Notes*, vol. 5, p. 27, 2012.

[113] Y. Cui, X. Liao, X. Zhu, B. Wang, and S. Peng, "mBWA: A Massively Parallel Sequence Reads Aligner," in *8th Int. Conference on Practical Applications of Computational Biology & Bioinformatics*, ser. Advances in Intelligent Systems and Computing, 2014, vol. 294, pp. 113–120.

[114] L. You and C. Congdon, "Building and Optimizing BWA ALN 0.5.10 for Intel Xeon Phi Coprocessors," https://github.com/intel-mic/bwa-aln-xeon-phi-0.5.10, [Online; accessed May, 2015].

[115] R. Luo, J. Cheung, E. Wu, H. Wang, S.-H. Chan, W.-C. Law, G. He, C. Yu, C.-M. Liu, D. Zhou, Y. Li, R. Li, J. Wang, X. Zhu, S. Peng, and T.-W. Lam, "MICA: A fast short-read aligner that takes full advantage of Many Integrated Core Architecture (MIC)," *BMC Bioinformatics*, vol. 17, p. 7, 2015.

[116] "Apache Cassandra home page," http://cassandra.apache.org/, [Online; accessed February, 2016].

[117] "Apache Parquet home page," http://parquet.apache.org/, [Online; accessed February, 2016].

[118] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[119] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Hardware Acceleration of Genetic Sequence Alignment," *Reconfigurable Computing: Architectures, Tools and Applications. Lecture Notes in Computer Science.*, vol. 7806, pp. 13–24, 2013.

[120] Y. Sogabe and T. Maruyama, "An acceleration method of short read mapping using FPGA," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 350–353.

[121] H. Waidyasooriya and M. Hariyama, "Hardware-Acceleration of Short-read Alignment Based on the Burrows-Wheeler Transform," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

[122] G. Zhao, C. Ling, and D. Sun, "SparkSW: Scalable Distributed Computing System for Large-Scale Biological Sequence Alignment," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015, pp. 845–852.

[123] T. J. Wheeler and J. D. Kececioglu, "Multiple alignment by aligning alignments," *Bioinformatics*, vol. 23, no. 13, pp. i559–i568, 2007.

[124] M. N. Price, P. S. Dehal, and A. P. Arkin, "FastTree2 – approximately maximum-likelihood trees for large alignments," *PloS one*, vol. 5, no. 3, p. e9490, 2010.

[125] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS'67 (Spring)*, 1967, pp. 483–485.

[126] S. R. Eddy *et al.*, "A new generation of homology search tools based on probabilistic inference," in *Genome Inform*, vol. 23, no. 1, 2009, pp. 205–211.

[127] R. D. Finn, J. Clements, and S. R. Eddy, "HMMER web server: interactive sequence similarity searching," *Nucleic acids research*, p. gkr367, 2011.

[128] D. Beazley, "Understanding the Python GIL," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[129] S. Guo, L.-S. Wang, and J. Kim, "Large-scale simulation of RNA macroevolution by an energy-dependent fitness model," *ArXiv e-prints*, Dec. 2009.

[130] J. J. Cannone, S. Subramanian, M. N. Schnare, J. R. Collett, L. M. D'Souza, Y. Du, B. Feng, N. Lin, L. V. Madabusi, K. M. Müller, N. Pande, Z. Shang, N. Yu, and R. R. Gutell, "The Comparative RNA Web (CRW) Site: an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs," *BMC Bioinformatics*, vol. 3, no. 1, p. 2, 2002.

List of Figures

List of Tables