CÁLCULO DE LA LIBRERÍA EN HADOOP Y SU INTEGRACIÓN EN T_COFFEE

Introducción al Proyecto

Autor:
Oscar Ujaque Perez

Directores:
Dr. Fernando Cores Prado
Dr. Jordi Lladós Segura

Máster en Ingeniería Informática
Universidad de Lleida
Septiembre 2017

Agradecimientos

Me gustaría agradecer a los directores del proyecto, el Dr. Fernando Cores y el Dr. Jordi Lladós, la incesable ayuda prestada en el desarrollo del mismo así como todos sus consejos, permitiendo culminar un trabajo que, en algunos momentos, llegó a parecer inalcanzable.

Quisiera agradecer al Dr. Fernando Guirado y demás docentes, todo lo que me han enseñado a lo largo de este proyecto y del Máster, haciendo de mí una persona un poco menos ignorante.

También me gustaría agradecer a mis compañeros de Máster, en especial a Mariano, todos esos ratos compartidos realizando prácticas y demás. Ratos que me han curtido un poco más como persona.

Finalmente, quisiera agradecer a mi familia y en especial a mi pareja Eva todo el apoyo que siempre me han brindado.

A todos ellos, ¡Gracias!

Índice

1.JUSTIFICACIÓN

Los alineadores múltiples de secuencias (MSA por sus siglas en inglés) son unas herramientas bioinformáticas, software básicamente, que consisten en una forma de representar y comparar dos o más secuencias (entre las que se comprenden el ARN, el ADN y algunas estructuras primarias proteicas) y cuya principal función es resaltar las zonas de similitud entre ellas. Estas zonas de similitud sirven para encontrar razones evolutivas o funcionales entre las secuencias, hecho de considerable importancia a nivel biológico.

Con la irrupción de las nuevas tecnologías de secuenciación, tenemos que cada vez las secuencias a alinear son más numerosas y más grandes. El aumento del número de secuencias y su longitud implican un crecimiento exponencial del tiempo requerido para alinearlas. Esto representa un importante limitación para la escalabilidad y aplicabilidad de estas herramientas. Por tanto, desde la vertiente más informática y para mejorar el rendimiento de estas herramientas, se necesita que los MSA sean capaces de aprovechar los recursos distribuidos de los grandes supercomputadores.

En este trabajo se va a investigar sobre una herramienta MSA denominada T_Coffee (Tree-based Consistency Objective Function For alignment Evaluation), que ha sido desarrollada en el Centro de Regulación Genómica (CRG) y está clasificado como uno de los mejores algoritmos de alineamiento de la última década. T_Coffee es un software para alineamientos de secuencias basado en un enfoque progresivo y la utilización de la consistencia.

El alineamiento progresivo consiste en ir alineando las secuencias de forma progresiva por pares siguiendo un árbol guía. Es decir primero alínea las secuencias más parecidas y posteriormente va añadiendo al resultado del alineamiento anterior a la siguiente más semejante.

Por otro lado, la librería de consistencia se construye a partir de la construcción de una librería de pares de alineamientos globales y locales. El objetivo de esta librería es minimizar los errores que se producen en las primeras fases de alineamiento por pares incluyendo información de consistencia referente a los alineamientos que se realizarán en etapas posteriores.

Además del problema señalado del alto coste computacional al alinear, cuando el número de secuencias es elevado (pocos cientos), T_Coffee también se ve afectado por los grandes requisitos de memoria necesarios para gestionar la librería de consistencia. Este problema viene dado porque T_Coffee guarda en memoria la librería primaria y, para un conjunto de muchas secuencias, la cantidad de memoria que necesita es tan elevada que la máquina que lo ejecuta se queda sin recursos.

En general, T_Coffee produce una alineamiento mejor que muchos de los métodos basados en la estrategia progresiva. Sin embargo, la mejoras que hacen que T_Coffee sea mejor, penalizan a T_Coffee en su velocidad y provoca que sea más lento que las otras

alternativas. A continuación se presentan diversos gráficos que presentan los órdenes de magnitud necesarios por T_Coffee para generar la librería y alinear distintos datasets.

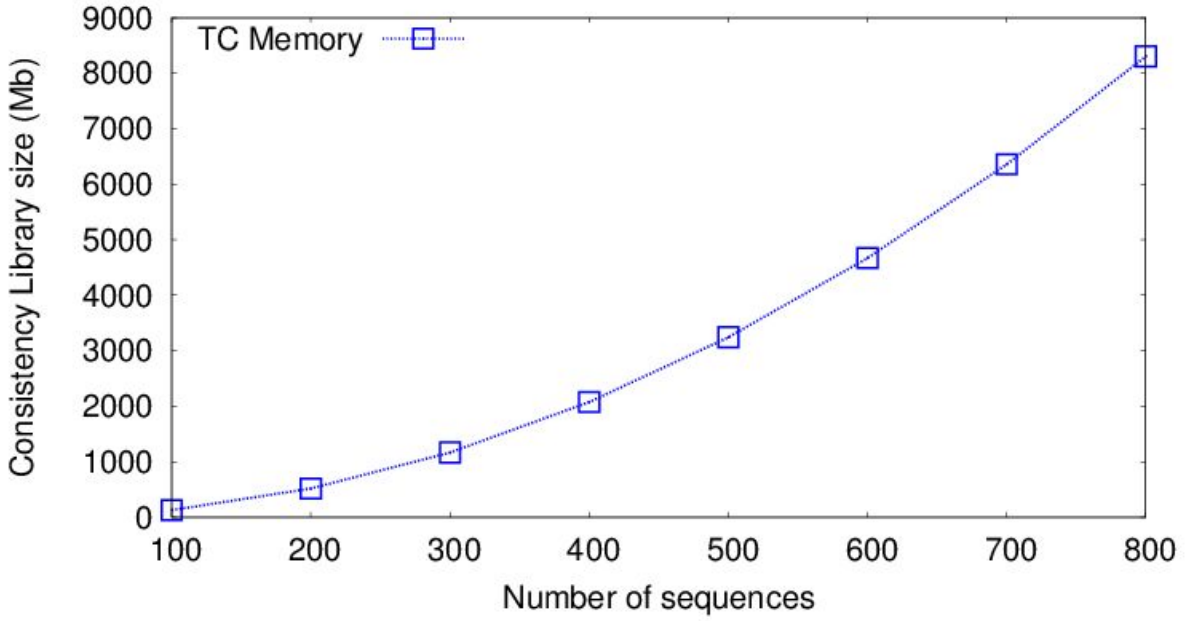

Figura 1: Memoria consumida por T_Cofffee en función del número de secuencias.

En la Figura 1 podemos observar cual es el espacio consumido a medida que aumentan el número de secuencias a alinear. Así se puede observar que para alinear 100 secuencias, T_Coffee utiliza una media de 129 MB mientras que para alinear datasets de 800 secuencias utiliza más de 8GB.

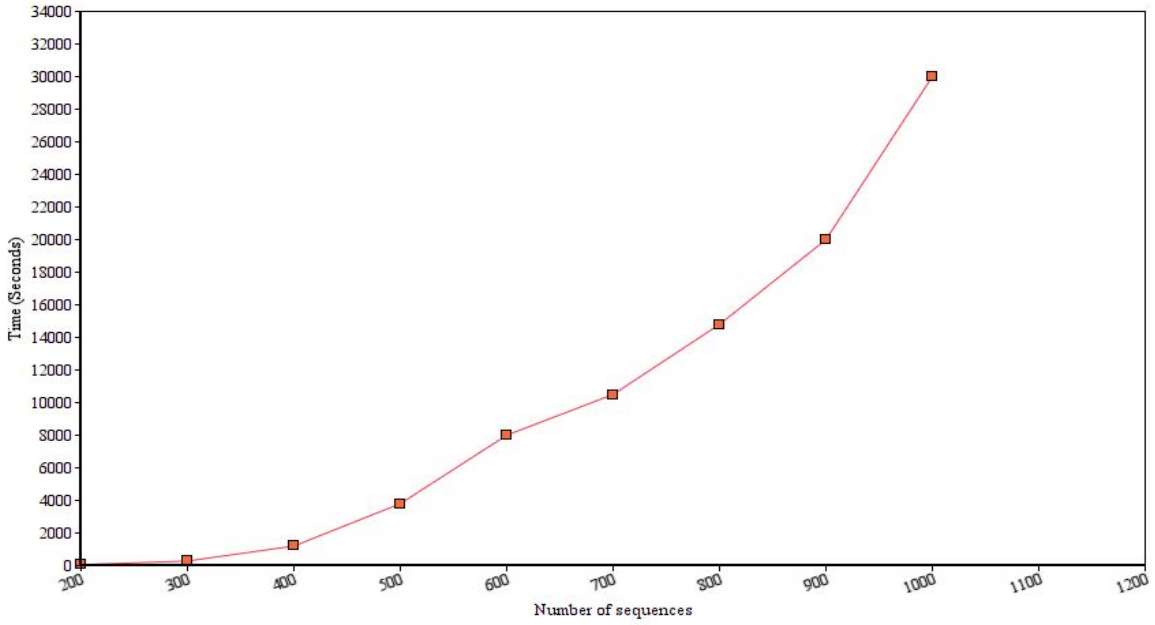

Figura 2: Tiempo utilizado por T_Coffee para alinear

En la Figura 2 observamos que el tiempo de ejecución de T_Coffee aumenta cuadráticamente en función del número de secuencias. Así, para alinear un dataset de 100 secuencias toma alrededor de 1 minuto mientras que para alinear datasets de 800 toma más de 8 horas, unas 380 veces más.

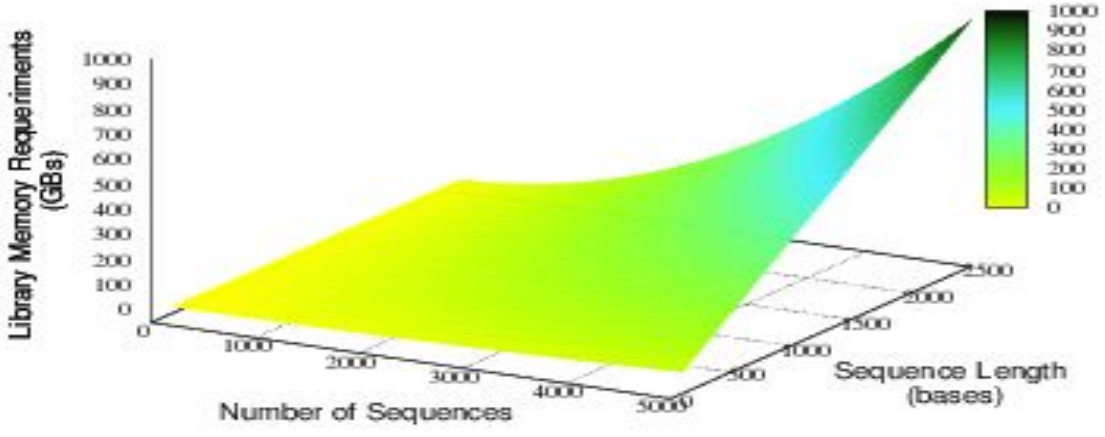

Figura 3: Memoria necesaria por T_Coffee en función del número y longitud de secuencias.

En la Figura 3 observamos la relación de memoria necesaria en función del número y longitud de las secuencias a alinear. Vemos que para datasets de 5000 secuencias y de una longitud de un 2500 caracteres el espacio necesario se eleva a 1000GB (1 Terabyte). Cabe destacar que los resultados analíticos de la gráfica se han obtenido a partir de la modelización de los requisitos de memoria de T_Coffee.

Así, para solventar el problema de memoria proponemos adaptar T-Coffee para que sea capaz de utilizar los recursos de las grandes infraestructuras Big Data. Migrando T-Coffee a un entorno hadoop nos permite solventar parcialmente los dos grandes problemas de T-Coffee: sus grandes requisitos computacionales y de memoria. Los requisitos de cómputo se solucionan gracias a la utilización de Spark, que es capaz de utilizar los recursos distribuidos de un cluster hadoop para calcular de forma distribuida la librería de consistencia. Respecto a los requisitos de memoria para la librería se pueden soslayar utilizando el almacenamiento en disco (que es varios órdenes de magnitud mayor que la memoria) de una base de datos distribuida como Cassandra para almacenar y acceder a la librería de consistencia. Está variante de T_Coffee, la hemos llamado SCoffee (Scalable T-Coffee).

A grandes rasgos, SCoffee hace uso de Spark para generar la librería primaria y de Cassandra para almacenarla (en vez de usar la memoria de la máquina). Para realizar el alineamiento, SCoffee hace llamadas a la base de datos para saber los datos de la librería primaria (en vez de consultar en memoria como la hace T_Coffee).

Este procedimiento tiene diversas ventajas, entre las que se encuentra, el reducir el problema de memoria, ya que Cassandra almacena datos en disco. Esto permite que SCoffee pueda alinear cualquier conjunto de secuencias siempre que su librería quepa en el almacenamiento distribuido (HDFS) del cluster de Big Data que se esté utilizando.

Además, el uso de Spark como herramienta para generar la librería primaria y almacenarla en Cassandra, tiene la ventaja de utilizar el procesamiento paralelo inherente a la filosofía hadoop/Spark, permitiendo incrementar las prestaciones de la aplicación, escalando la infraestructura con la adición de nuevos nodos al cluster hadoop. Este hecho nos permite reducir el tiempo de generación de la librería primaria solo con el hecho de incrementar el número de nodos utilizados.

2.OBJETIVOS

El objetivo principal, a parte de la realización del TFM, es construir un herramienta informática que permita el alineamiento de un conjunto con elevado número de secuencias mejorando la escalabilidad de T_Coffee.

Otro objetivo es mejorar el tiempo total empleado en el alineamiento gracias al uso de Spark. Con Spark se puede mejorar el tiempo de construcción de la librería primaria con el simple hecho de añadir más nodos al cálculo. Por lo tanto, y como la librería primaria es estrictamente necesaria para el alineamiento, se puede reducir el tiempo total de cómputo de la librería primaria y por lo tanto de alineamiento en general.

Por otro lado, Spark guarda la librería primaria en Cassandra para evitar los problemas de memoria. Por lo tanto, SCoffee deberá acceder a Cassandra para obtener los datos de la librería primaria e ir construyendo el alineamiento. Este hecho penaliza a SCoffee en tiempo, ya que acceder a Cassandra siempre será más lento que acceder a memoria (como lo hace T_Coffee).

Por lo tanto, otro objetivo del proyecto es optimizar el tiempo de alineamiento de SCoffee. Dado que el tiempo de acceso a la base de datos Cassandra no se puede modificar, se han utilizado diversas técnicas para minimizar las consultas a Cassandra.

Estas nuevas estructuras son una caché de consistencia, un nuevo concepto chunk para agrupar datos en Cassandra y reducir los accesos a la base de datos y unas políticas de reemplazo (LRU y Random), que reducen al máximo las peticiones a Cassandra (estos nuevos conceptos serán explicados más en detalle en apartados posteriores).

3.PLANIFICACIÓN

En la siguiente imagen se puede encontrar una temporización del proyecto o diagrama de Gantt que muestra las tareas realizadas y el tiempo empleado desde el inicio hasta la conclusión del proyecto:

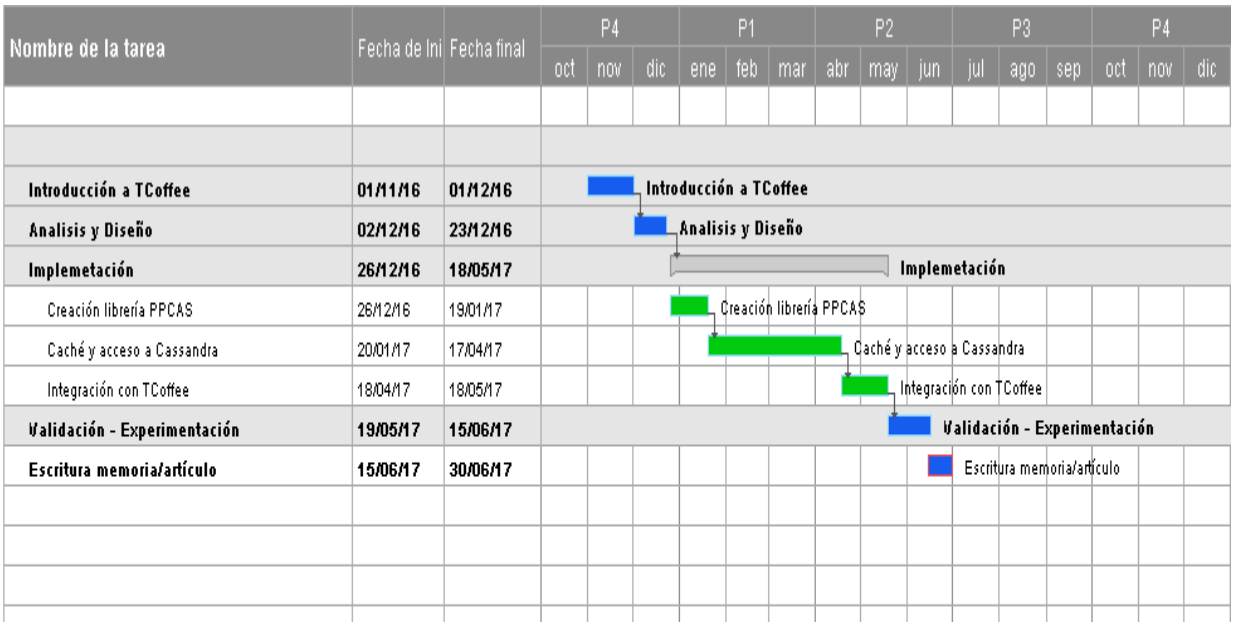

Figura 4. Diagrama de Gantt

Las tareas a realizar serán las siguientes:

Introducción al T_Coffee:

La primera tarea consistirá en familiarizarse con T_Coffee. Comprender en qué consiste y cuál es su función. Familiarizarse con el código y comprender a grandes rasgos los objetivos del proyecto.

Análisis y Diseño

La segunda tarea consistirá en analizar el proyecto. Comprenderá analizar los requisitos que serán necesarios para su posterior implementación. Se prevé una metodología hacia la vertiente ágil, debido a que la implementación y soluciones se desarrollarán a partir de la necesidad de proyecto y de los resultados parciales obtenidos.

Implementación

Esta fase implicará todo el desarrollo del código. Se puede dividir en tres subtareas:

- Creación de la librería PPCAS: En esta subfase se implementará la librería PPCAS. Esto implica desarrollar la parte que, dado el código que genera la librería primaria, permita insertarla en Cassandra.

- Caché y acceso a Cassandra: En esta subfase se implementará el acceso a Cassandra para para descargar la librería primaria que permita a SCoffee generar el alineamiento. También se desarrollará la caché que permitirá guardar en memoria la librería primaria descargada de Cassandra.

- Integración con T_Coffee: En esta subfase se integrará todo el código explicado anteriormente, de manera que, desde la línea de comandos de T_Coffee se pueda realizar todo el proceso (generar librería con Spark y guardarla en Cassandra y realizar el alineamiento mediante el uso de la librería descargada desde Cassandra).

Validación y Experimentación:

En esta fase se realizará el testeo del código que implicará probar la nueva implementación con diversos conjuntos de secuencias y observar cómo se comporta. También implicará analizar dichos resultados y contrastar los resultados obtenidos con los que genera T_Coffee.

Escritura memoria/artículo:

Una vez realizadas todas la pruebas y testeado el código se escribirá el artículo y la introducción al proyecto.

4.ANÁLISIS Y DISEÑO DE SCOFEE

Tal como ha se ha comentado anteriormente SCoffee utiliza Spark, Cassandra y T_Coffee. En la Figura 2 puede ver al interacción de todos ellos.

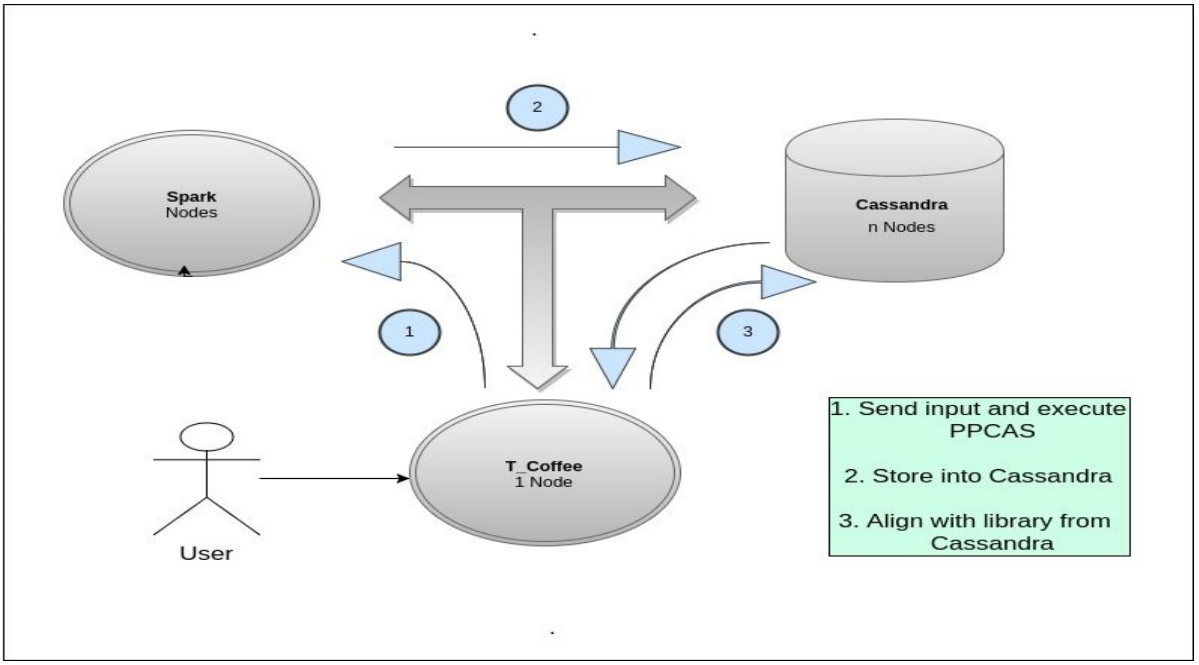

Figura 5: Diagrama de flujo de SCoffee

Como se puede observar en la Figura 5, el usuario ejecuta SCoffee desde una máquina. Esta máquina a su vez, y en primera instancia, enviará el input (o dataset inicial) a los nodos de Spark y ejecutará PPCAS.

Posteriormente, mediante el paradigma *Map-Reduce* (se hace un *map* a la función principal de PPCAS mediante código Python) se calcula la librería primaria y se almacena en Cassandra.

Finalmente, cada vez que a SCoffee le hacen falta datos de la librería primaria para alinear, hace una consulta a Cassandra mediante *SELECT's*. Por lo tanto, aquí se crea el flujo de datos entre T_Coffee y Cassandra.

4.1. Requisitos:

Los principales requisitos de SCoffee son:

- Una caché de consistencia donde se guardará toda la librería primaria bajada de Cassandra. Este caché se utiliza para evitar un gran número de peticiones a Cassandra ya que T_Coffee llama repetidamente por los mismos datos de la librería.

- Unas políticas de reemplazo de la caché de consistencia ya que la memoria RAM del equipo (que es la que utiliza la caché) donde se ejecuta el programa es limitada.
- Un pool de threads que permite hacer peticiones concurrentes a la base de datos y a su vez permite bajarse la librería primaria en adelanto.
- Una política de almacenamiento en la base de datos para optimizar la forma de tratar la librería una vez descargada de la base de datos.
- Una reducción del impacto de la latencia al traer datos de Cassandra mediante el uso del chunk que permite aumentar el tamaño de la información traída de la base de datos.

4.2. Diseño:

4.2.1 Cache de Consistencia:

Dado el hecho que T_Coffee llama repetidas veces a los mismos datos relacionados a una Secuencia y a su residuo, y para evitar idénticas peticiones a Cassandra, se ha usado un triple puntero de enteros (o caché de consistencia -CC-) Esta caché de consistencia guarda los datos que son obtenidos en una consulta a Cassandra. Así pues, para la segunda o posteriores veces que T_Coffee pida esos datos, ya estarán guardados en la CC. Por lo tanto, para esas posteriores veces que se necesitan los datos se retornan los datos almacenados en la CC. Siempre resultará ser más rápido consultar a memoria (a la CC) que consultar a la base de datos.

En la Figura 6 se puede ver la estructura de la CC. Se observa que en su primera dimensión está compuesta por la Secuencia (Seq N), en la segunda dimensión está compuesta por el Residuo (Res N) y en su tercera dimensión está compuesta por diferentes Secuencias (Seq i) y Residuos (Res i) y su peso asociado (Weight i). Cabe destacar que la primera posición de la tercera dimensión alberga la longitud de ese vector.

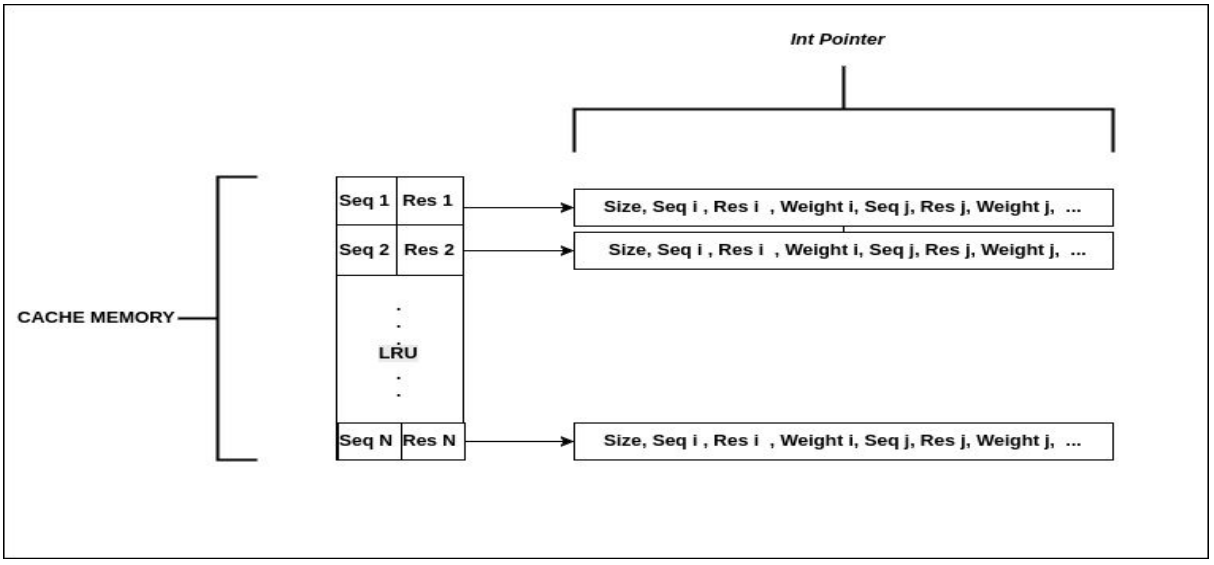

Figura 6: Estructura de la caché de consistencia

4.2.2 Políticas LRU (Least Recently Used) y Random:

Para mejorar el tiempo de ejecución se utiliza la CC, pero ésta a su vez consume memoria de la máquina que a su vez es limitada. Por lo tanto, es necesario llevar un control de la memoria usada por el programa.

Necesitamos limitar la memoria utilizada por la caché de consistencia para que no supere los recursos de memoria disponible en la máquina. Para ello, si no hay espacio en memoria/caché suficiente para ubicar la últimos residuos accedidos, entonces es necesario lograr un hueco en la caché. Para ello, utilizaremos las políticas de reemplazo.

Para llevar ese control de memoria se utilizan diferentes funciones (que se explicarán en el apartado implementación). Así pues, una vez llegados al límite de memoria RAM consumible por el programa se han de definir unas políticas de reemplazo de la caché.

- LRU (Least Recently Used):

  La LRU es un hash que va albergando elementos (pares secuencia-residuo en este caso). Cuando se quiere insertar un elemento y ya está dentro, este elemento es desplazado a la cola del hash.

  Cada vez que T_Coffee pregunta por datos relacionados a un par secuencia-residuo, este par se guarda en la LRU (según la forma explicada anteriormente). Cuando la memoria consumida por la CC llega a su límite, hay que liberar memoria. La forma de recuperar memoria en el sistema es liberar el la tercera dimensión (o vector) que tienen por coordenadas el par secuencia-residuo que se extrae de la LRU. Este par que se extrae es el frontal de la LRU que es a su vez el par secuencia-residuo que hace menos tiempo que se usa.

  Esta política de reemplazo de los elementos de la CC permite vaciar elementos que probablemente no serán necesitados más por el programa. Este hecho es debido a que T_Coffee puede llamar muchas veces seguidas a datos relacionados a un par secuencia-residuo y los suele llamar seguidamente.

- Random:

  La política Random es una política de reemplazo de memoria aleatoria. Es decir, elige aleatoriamente un par secuencia-residuo y si su tercera dimensión (o vector) no está vacío lo libera. El objetivo de esta política es validar la efectividad de la política de reemplazo LRU y demostrar la localidad de los accesos a la librería de consistencia

  A grandes rasgos, estas dos políticas se aplican mientras la memoria consumida calculada teóricamente continúa siendo mayor a la que puede utilizar el programa o a una memoria designada por parámetro.

4.2.3 Chunk. Organización y estructura de datos en Cassandra

Cuando SCoffee debe acceder a la librería primaria, y para evitar hacer muchas consultas a Cassandra, se utiliza un nuevo concepto *chunk*. Este *chunk* es un entero que se pasa por parámetro a SCoffee que permite agrupar datos en la base de datos optimizando el número de accesos a la misma. Además permite llevar un control de cómo se almacenan los datos en Cassandra y de cómo se recuperan. A continuación se presenta cómo se hace uso del *chunk* y cómo se organizan los datos en Cassandra.

Dado que Cassandra es una base de datos clave-valor, se ha utilizado la siguiente forma para almacenar (que a su vez es la óptima para tratar los datos una vez recibidos en T_Coffee desde Cassandra):

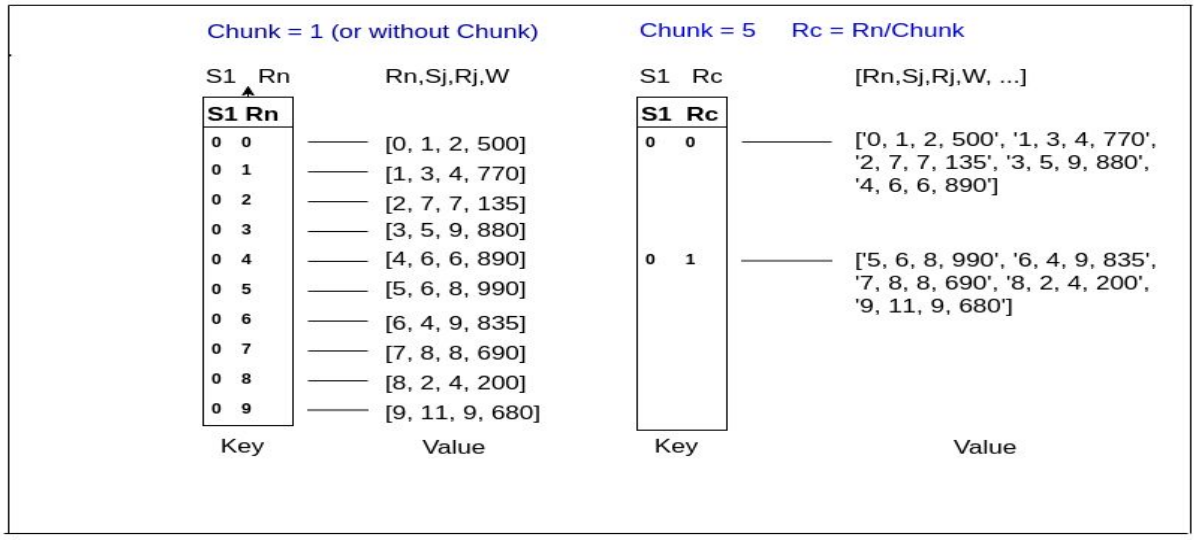

Figura 7 : Distribución en Cassandra según el chunk

En la Figura 7 observamos dos formas de distribuir los datos. A la izquierda se observa la distribución con chunk 1 (lo que sería equivalente a no tener chunk) donde vemos que para cada clave (secuencia-residuo) se guardan los datos relacionados a la otra secuencia-residuo y el peso. En la parte derecha se observa una distribución con chunk 5 donde se ve que la clave viene dada por la secuencia y el residuo dividido por el chunk y en los valores se agrupan diferentes datos relacionados a diferentes secuencias y residuos y pesos tales que la división entera del residuo (Rn) por el chunk es la misma. Mientras que en la tabla de la izquierda hay diez columnas que implican diez accesos en la tabla de la derecha hay dos columnas que implican dos accesos a la base de datos.

Esta forma de guardar datos en Cassandra permite guardar muchos datos para cada par secuencia-residuo, ya que el residuo viene dividido por el chunk. Por lo tanto, también permite que después, cuando SCoffee pide los datos relacionados a esa secuencia-residuo, se bajen todos los datos relacionados a esa secuencia-residuo/chunk (y se guarden en la caché de consistencia) con lo que se evita hacer múltiples peticiones a Cassandra y por lo tanto ganar tiempo.

Para dar más claridad al concepto *chunk* se presenta a continuación el comando que carga los datos en Cassandra (en PPCAS):

UPDATE TABLE SET VALUE=VALUE+[**'R1, S2, R2, W12'**] WHERE SEQ = **S1** AND RES = **R1/CHUNK**

donde SEQ y RES son la clave de la tabla.

Se puede observar que se suben a la misma clave todos los datos relacionados a los residuos que tienen igual parte entera de la división R1/CHUNK.

4.2.4 Pool de threads.

Para aumentar la velocidad de alineamiento de SCoffee se ha usado un pool de *threads*. Este pool empieza a hacer las peticiones a Cassandra y empieza a llenar la caché de consistencia justo cuando se ejecuta el programa. El pool va haciendo peticiones a Cassandra (de todos los posibles pares secuencias-residuos que existan) mientras el programa va haciendo otras tareas.

En futuras implementaciones el pool de threads también permitirá traer en adelanto información a la caché, siguiendo el orden de alineamiento definido por el árbol guía.

5.IMPLEMENTACIÓN

5.1 Creación de la librería de consistencia en Cassandra (PPCAS+Cassandra).

Dado que la tecnología a usar era Spark (que solo admite Java, Python o Scala como lenguajes de programación), se probó de optimizar unas traducciones, ya dadas, del código fuente (en C) a Python. Para eso se estuvieron probando diversas optimizaciones de código, entre las que se encuentran el Jit Compiler que sirve para compilar en tiempo real. Aún así no se obtuvieron resultados cerca de lo esperado por lo que se descartó.

Aun con el uso de compiladores ***just-in-time***, las traducciones de código a Python eran más lentas que el código C puro. Este hecho nos obligó a la búsqueda de otras soluciones que permitieran obtener velocidades, si más no, similares a las de C.

Finalmente, se estudió cómo envolver código C para que pudiera ser utilizado con Python. Para eso se estudiaron brevemente algunas metodologías y se acabó eligiendo Ctypes. Ctypes permitía usar código C desde Python, con las ventajas que eso conlleva (la velocidad de C y la sencillez de Python).

Después vino la integración de PPCAS (parte de código que genera la librería primaria) con Cassandra. Para ello, una vez estudiados las conexiones entre C y Cassandra, se desarrolló una metodología de manera que, cuando T_Coffee generara la librería, ésta se insertara en Cassandra.

Una vez desarrollada esta parte, se procedió al siguiente punto que era realizar las peticiones desde SCoffee a Cassandra para ir realizando el alineamiento.

Una vez desarrollado el código para ambas tareas anteriores, introducir librería primaria en Cassandra con Spark y obtener librería de Cassandra con T_Coffee, llegó la parte de integración de todos ellas en SCoffee. La idea fue, desarrollar SCoffee de manera que, desde donde se invoca a T_Coffee se pudieran añadir parámetros para invocar Spark, generar librería y también guardarla en Cassandra.

Además SCoffee se implementó de manera que Spark, T_Coffee y Cassandra se pudieran ejecutar en máquinas diferentes. Así, desde la máquina en que se ejecuta T_Coffee y mediante *ssh* se puede conectar a la máquina que ejecuta el driver de Spark. A su vez, éste mediante las conexiones a Cassandra permite insertar datos en otra máquina diferente donde se encuentra el máster de la base de datos. De esta forma permite distribuir las tres tecnologías en diferentes máquinas y aprovechar al máximo los recursos de cada una.

Inserción en Cassandra (PPCAS + Cassandra)

SCoffee, en la parte de almacenar la librería primaria en Cassandra, está compuesta por dos archivos principales:

- El primer archivo es PPCAS.c. Este archivo contiene las funciones de T_Coffee, que dado un conjunto inicial de secuencias, genera la librería primaria. A parte, PPCAS.c contiene una modificación que permite guardar la librería primaria en Cassandra.

- El segundo archivo es PPCAS.py. Este archivo contiene el código Python/Spark con el que se aplica el paradigma ***map/reduce***. Este ***map***, que se aplica en la función principal de PPCAS.c, distribuye el cálculo de la librería en diferentes tareas Spark que son lanzadas y ejecutadas en distintos nodos del clúster hadoop. También cabe destacar, que para poder usar PPCAS.c en PPCAS.pỳ el archivo PPCAS.c es préviamente envuelto con Ctypes.

Por lo tanto, para implementar la parte que inserta librería primaria en Cassandra se han añadido diversas funciones al código original de T_Coffee (obteniendo PPCAS.c como resultado). Estas funciones son las siguientes:

- ***int proba_pair_wise (char* s1 , char * s2, int seq1 , int seq2, char * c _ip, char *out name, int to _file, int *lib, int chunk);***

  Esta función, y dependiendo del parámetro to_file, puede llamar a la función ProbaMatrix2CL o a la función ProbaMatrix2CL_CASS. Si se llama a ProbaMatrix2CL se ejecuta el procedimiento normal de T_Coffee (en memoria). Si llama a ProbaMatrix2CL_CASS significa que la parte de insertar datos en Cassandra se activa.

La función ProbaMatrix2CL_CASS tiene la siguiente estructura:

- ***int ProbaMatrix2CL_CASS (int I, int J, int NumMatrixTypes, int NumInsert-States, float *forward, float *backward, float thr, int *lib, int seq1, int seq2, char *c_ip, char * out_name, int chunk);***

  Esta función alberga diferentes funciones. Inicialmente, hace la conexión a Cassandra mediante el uso de dos funciones *create_cluster* y *connect_session*. Mediante *addbatch y la* función *execute_batch* carga los datos en Cassandra. Y finalmente, mediante la funcion *create_schema* comprueba si se ha creado el *keyspace* en Cassandra (si el *keyspace* no está creado, lo crea) y comprueba si la tabla donde se van a insertar los datos existe (si la tabla existe la borra y crea una de nueva y si no existe solamente la crea).

Las funciones anidadas dentro de la función *ProbaMatrix2CL_CASS d*escritas anteriormente son las siguientes:

- ***CassCluster * create_cluster(const char* hosts);***

  Esta función crea la configuración para conectar con Cassandra.

- ***CassError connect_session(const CassCluster * cluster);***

  Esta función realiza la conexión a Cassandra.

- ***add_batch(CassBatch * batch, int seq1, int seq2, char * out_name, int r1, int r2, int w, int chunk, char * c_ip);***

  Esta función, principalmente añade los *UPDATES* (previamente comentados) a un *batch.*

- ***void execute batch(CassBatch* batch);***

  Esta función ejecuta el batch. Por lo tanto, es la que inserta los datos en la tabla de Cassandra.

- ***create_schema(char * c ip, char * out_name);***

  Esta función comprueba que no existan el *keyspace* y la tabla. Si es así, los crea.

La parte de código de Python (PPCAS.py) primero de todo, comprueba que existan el *keyspace* y la tabla llamando a la función *create_schema* (envuelta con CTypes). Tal como se ha explicado anteriormente, si no existen, los crea.

Después del paso anterior, PPCAS.py aplica el paradigma *map* a la función *proba_pair_wise,* (que es la que está más arriba en jerarquía de las funciones comentadas anteriormente) que hace un mapeo a todos los elementos contenidos en el RDD de Spark.

Todas esta funciones se utilizan en el código Python mediante un archivo PPCAS.so que es el PPCAS.c envuelto por CTypes listo para ser usado des del código Python.

5.2 Consultas a Casandra (T_Coffee + Cassandra)

En el archivo tcoffee_cassandra.c se pueden encontrar todas las funciones que se han implementado y que encargan de realizar las consultas a Cassandra.

Entre las funciones de tcoffee_cassandra.c podemos encontrar algunas que son ejecutadas en el batch_main.c de T_Coffee y otras que son ejecutadas desde el archivo evaluate.c.

En el primer conjunto de funciones encontramos (las ejecutadas en el batch_main.c):

- ***void fillConstraint_list(Constraint_list **CL,int chunk_cass, int pid, char *argv2, char * cass_table, int memory, char* masterip, char* cassandraip, int mru, int n core, std::string * s2, std::string * tableUsed);***

Esta función, principalmente, se encarga de rellenar la CL (estructura Constraint_list de T_Coffee) con los nuevos parámetros que se pasan como argumento en el comando que ejecuta T_Coffee. De esta forma los nuevos parámetros son accesibles desde cualquier parte del programa.

La siguiente función es:

- ***void enableSpark(int spark, const char * cassandraip, char * masterip, int chunk_cass, char *table, int partitions, std::string dataset);***

  El principal objetivo de esta función es lanzar Spark desde T_Coffee. Si un parámetro del comando que ejecuta T_Coffee está activado (*spark*) se ejecuta Spark con todos los parámetros que se pasan a la función.

Otras funciones llamadas desde el *batch_main*.c son las siguientes:

- ***void enablePool(int pool, Constraint list ** CL, const char * cassandraip, intchunk cass);***

  El principal objetivo de esta función es activar el *pool de threads* cuando T_Coffee se empieza a ejecutar. Si se activa un parámetro en el comando principal (*pool*), tantos *threads* como indique el parámetro empiezan a hacer peticiones a Cassandra para ir llenando la CC.

void dropTable (int remove_table, char * table, Constraint_list * CL);

Esta función borra permanentemente de la base de datos (una vez concluído el alineamiento) la tabla creada por la parte de código Python (PPCAS.py) y que a su vez, está llena de datos. Se activa dependiendo del un parámetro pasado por argumento (remove_table).

En el segundo conjunto de funciones, las que son llamadas desde evaluate.c encontramos las siguientes:

- ***void updateMemory(pid t pid);***

  y

- ***void ManagmentSigchld(int s);***

  Estas dos funciones se utilizan para saber cuándo muere un proceso utilizando la señal SIGCHILD para recibir la notificación. SCoffee, igual que T_Coffee, permite la ejecución en modo multiproceso. Por lo tanto, cada vez que muere un proceso es necesario saberlo para resetear a 0 la memoria RAM teórica consumida por ese

proceso. De esta forma, se puede llevar, en cada momento, un control casi exacto de la memoria total consumida por el programa.

Otra función utilizada es:

- ***int initialMem(Constraint_list * CL);***

  Esta función calcula teóricamente la cantidad de memoria que consume SCoffee al iniciarse. Básicamente calcula cuál es la memoria consumida por la caché de consistencia cuando está inicializada.

La función más importante de este fichero es la siguiente:

- ***int * query(int seq, int res, Constraint_list * CL, int pid);***

  Esta función primero de todo, crea un vector de memoria compartida entre todos los procesos. Con este vector (indexado por *pid*) se puede saber, tan solo sumando la posiciones del vector que tiene el *pid* igual al de los procesos que la ejecutan, cuál es la memoria total consumida por cada proceso y por ende la memoria total consumida por el programa.

  Posteriormente, la función hace la consulta a Cassandra y descarga los datos. Si el chunk es mayor que uno, se utiliza la técnica del chunk y baja todos los datos relacionados al par secuencia-residuo/chunk.

  El siguiente paso es comprobar que la memoria calculada teórica (que se suele ajustar bastante bien a la real consumida) no sea mayor que la disponible por la máquina (o por una cantidad pasada por parámetro al comando de ejecución de SCoffee). Si la memoria consumida teórica es mayor que la disponible (o la pasada por parámetro) empieza a reemplazar memoria de la caché de consistencia. Este reemplazo de memoria se puede hacer mediante dos métodos, el método LRU o el método Random (que serán explicados posteriormente).

  A continuación, cuenta cuántos elementos, de cada par secuencia-residuo, se ha bajado de la base de datos (mediante la técnica del chunk). Con el valor que obtiene reserva el espacio necesario para el puntero de la tercera dimensión de la caché de consistencia.

  El último paso de esta función, es poner todos los elementos descargados de Cassandra mediante la técnica del chunk en un iterador. Una vez puestos en el iterador, son recorridos para ser insertados en la caché de consistencia.

Los dos métodos de reemplazo de memoria tienen la siguiente definición:

- ***void deleteLru(Constraint_list * CL, int pid);***

Esta primera función libera memoria RAM mediante el uso de la LRU (previamente definida). Cuando hace falta memoria, la LRU borra el elemento ubicado en el frontal del hash que la define y libera el espacio designado, por esas coordenadas, de la tercera dimensión de la caché de consistencia (el elemento borrado de la LRU tiene la forma de par secuencia-residuo).

Hay que tener en cuenta que la misma función, para mantener un orden en el relleno de la caché de consistencia, cada vez que la LRU designa un par secuencia-residuo para que su vector sea borrado, también borra todos los pares secuencia-residuo incluidos en ese chunk.

- ***void deleteRandom (Constraint_list * CL, int pid);***

Esta segunda función utiliza una técnica aleatoria (*Random*) para elegir el par secuencia-residuo a borrar. La función busca en la caché de consistencia aleatoriamente algún par secuencia-residuo que no tenga vacío su tercera dimensión. Entonces libera el vector asociado a las coordenadas de es par secuencia-residuo en la caché de consistencia y también todos los vectores asociados a todos los pares secuencia-residuo de ese chunk.

Otra función utilizada en la parte de consultas a Cassadra es la siguiente:

- ***int * queryCass(int seq, int res, Constraint_list * CL);***

Esta función es la que es llamada por SCoffee para recibir los datos para hacer el alineamiento. El procedimiento que sigue es el siguiente: cuando SCoffee pregunta por la librería primaria asociada a la secuencia y residuo pasados por parámetros, consulta si están en la caché de consistencia. Si están, devuelve la tercera dimensión (o vector) de caché de consistencia con todos los datos. Si no están, hace una llamada a la función *query* (comentada anteriormente) para hacer la consulta a Cassandra y obtener los datos de la librería primaria.

Finalmente, la última función que es utilizada en esta parte es la siguiente:

- ***unsigned long getMem(Constraint_list *CL);***

Esta función calcula la memoria libre disponible de la máquina que alberga SCoffee. Si un parámetro *memory* es especificado el el comando inicial y está memoria especificada es menor que la disponible por el sistema la función devuelve la memoria especificada mediante el comando. Si es mayor, devuelve la memoria libre disponible por el sistema.

El uso de esta función se hace para saber de qué memoria dispone el sistema al iniciar el programa (o la especificada por parámetro) y así no superar los límites y permitir que el programa no se bloquee por falta de memoria.

5.3 Nuevos parámetros añadidos a SCoffee.

Para poder ejecutar SCoffee, es decir T_Coffee junto a Spark y Cassandra se han añadido algunos parámetros que se especifican a continuación:

-memory: especifica la memoria RAM que va a usar el programa. Si ésta es mayor que la disponible por la máquina o el parámetro no es especificado asigna la memoria RAM máxima disponible por la máquina.

-chunk_cass: especifica cuál va ser el chunk usado para insertar y descargar datos de Cassandra. Si éste no es especificado, se asigna por defecto el valor uno.

-pool: especifica el número de threads que van a ser usados por el pool. Si éste no es especificado o es especificado como 0, el pool no se activa.

-ppcas: si este argumento es especificado con el valor 1, se ejecuta Spark (insertando datos en Cassandra) y se realiza el alineamiento. Si es especificado con el valor dos, ejecuta Spark y tras realizar la inserción en Cassandra el programa acaba sin realizar el alineamiento. Si no es especificado o especificado con el valor 0, Spark no se ejecuta, sólo se realiza el alineamiento.

-lru: si este parámetro es especificado con el valor uno, se ejecuta la política LRU para el liberado de memoria. Si es especificado con el valor 0 ejecuta la política de reemplazo aleatoria. Si no es especificado, toma la política LRU por defecto.

-del_table: si este parámetro es especificado con el valor uno, el programa borra la tabla usada de la base de datos. Si no es especificado o es especificado con el valor 0 el programa no borra la tabla de la base de datos.

-master_ip: si el parámetro ***ppcas*** está activo (1 ó 2) indica que Spark se activa, entonces este parámetro debe ser especificado con la dirección IP del máster de Spark.

-cassandra_ip: si este parámetro es especificado debe contener la dirección IP de Cassandra. Si no se especifica asigna por defecto el valor de la dirección IP del máster de Spark.

-use_table: si este argumento es especificado con una tabla de Cassandra válida, el programa realiza el alineamiento con dicha tabla

-partitions: si este parámetro es especificado y ***ppcas*** está activo, Spark realiza la inserción en Cassandra con tantos *Jobs* como se han indicado como parámetro.

-user: si *ppcas* está activo y este parámetro está especificado con un usuario válido, el programa realiza una conexión ssh a la máquina donde está albergado Spark (para ejecutarlo) con dicho usuario.

-path: si este parámetro está activo el programa realiza un *scp* a dicha ruta para enviar la *dataset* inicial.

6. RESULTADOS:

A continuación se presentan los resultados que se han obtenido para encontrar la mejor configuración de parámetros para posteriormente realizar el alineamiento. En el último apartado se presenta la comparativa de tiempos obtenidos entre SCoffee y T_Coffee para diferentes datasets. Para todos los tiempos se remarca en negrita los mejores tiempos obtenidos.

La infraestructura utilizada has sido un clúster formado por 20 máquinas de 4 núcleos cada una con Intel Core 2 Quad a 2.4 GHz y 8GB de memoria RAM.
La distribución utilizada para las pruebas ha sido la siguiente:

- 15 máquinas (60 cores) para Spark
- 4 máquinas (16 cores) para Cassandra.
- 1 máquina (4 cores) para T_Coffee.

El conjunto de datasets utilizado pertenece a una família de datasets (Homfam) que provee grandes conjuntos de secuencias. De esta familia se han seleccionado 5 clases (Acetyltransf, rrm, rvp, sdr y zf-CCHH) para así poder evaluar los resultados.

6.1 Nodos Cassandra.

El propósito de las ejecuciones siguientes es encontrar la mejor configuración en tanto que número de máquinas con Cassandra.

A continuación se presenta la comparativa de tiempos obtenidos para una misma configuración y distintos nodos de Cassandra.

Parámetros por defecto:

- Secuencia: rrm_500
- Chunk: 10
- Política reemplazo: LRU (no procede porque cabe en memoria)
- Memoria máxima: 5000 MB
- Cores: 4
- Pool: 10
- CPU Spark: 60

4 nodos C*	3 nodos C*	2 nodos C*	1 nodos C*

Spark	TCoffee	Spark	TCoffee	Spark	TCoffee	Spark	TCoffee
10’25’’	7’43’’	12’05’’	7’37’’	13’33’’	7’39’’	20’10’’	7’40’’

18’08’’	19’42’’	21’12’’	27’50

Si se observan los resultados se ve que la mejor configuración se da para el máximo número de nodos para Cassandra. Se observa también que el tiempo mejora sustancialmente cuando se da la inserción de datos en Cassandra (la parte de Spark).

Este hecho es debido a que para la inserción en Cassandra se utilizan *batch* de 60 *updates* cada *batch*. Este número multiplicado por 60 cores implican 3600 instrucciones que debe procesar Cassandra por cada unidad de tiempo. Por lo tanto, a más nodos de Cassandra para distribuir la carga mejor rendimiento.

6.2. Chunk

El propósito de estas ejecuciones es encontrar la mejor configuración en tanto que parámetro chunk.

A continuación se presenta la comparativa de tiempos obtenidos para una misma configuración y distintos chunks.

Parámetros por defecto:

- Secuencia: rrm_500
- Nodos C*: 4 (elegimos este número porque es el que tiene mejor rendimiento según los tiempo obtenidos en la prueba anterior):
- Política reemplazo: LRU (no procede porque cabe en memoria)
- Memoria máxima: 5000 MB
- Cores: 4
- Pool: 10
- CPU Spark: 60

Chunk 1	Chunk 10	Chunk 100	Chunk 1000

Spark	TCoffee	Spark	TCoffee	Spark	TCoffee	Spark	TCoffee
11’28’’	8’09’’’	10’25’	7’43’’	11’41’’	9’64’’	11’31’’	9’45’’

19’37’’	**18’08’’**	21’35’’	21’16’’

Si observamos los resultados podemos ver que el mejor tiempo obtenido es para el chunk diez. Aquí no hay una relación clara para definir el mejor parámetro. El hecho de que sea 10 puede ser debido que a medida que aumenta el chunk desciende el número de accesos a la base de datos pero a su vez aumenta la latencia y el tiempo en almacenar todos los datos descargados en la caché. Por lo tanto el óptimo debe estar entre 1 y 100 que en este caso para las pruebas realizadas es 10.

6.3. Política de reemplazo

El propósito de estas ejecuciones es encontrar la mejor política de reemplazo de entre las dos que hay definidas.

A continuación se presenta la comparativa de tiempos obtenidos para una misma configuración y distintas políticas de reemplazo.

Parámetros por defecto:

- Secuencia: rrm_500
- Nodos C*: 4 (la mejor configuración según las pruebas realizadas)
- Chunk: 10 (la mejor configuración según las pruebas realizadas)
- Memoria máxima: 432 MB (75% del total)
- Cores: 1
- Memoria usada por el programa: 576 MB
- Pool : 0

LRU (Tiempo TCoffee)	Random (Tiempo TCoffee)
16’54’’	19’30’’

Si observamos los resultados se puede ver que la mejor política de reemplazo es la LRU. Este hecho ya se esperaba dado la naturaleza de T_Coffee que llama seguida y repetidamente por los datos de la misma secuencia-residuo. Por lo tanto, la política de reemplazo LRU tal como está definida permite optimizar el rendimiento de la aplicación.

6.4. Uso de memoria.

El propósito de estas ejecuciones es ver el impacto del uso de la caché en la construcción del alineamiento. Para ello se irá reduciendo el tamaño máximo de la cachè en cada ejecución.

A continuación se presenta la comparativa de tiempos obtenidos para una misma configuración y distintos tamaños de memoria caché.

Parámetros por defecto:

- Secuencia: rrm_500
- Nodos C*: 4 (mejor parámetro según las pruebas realizadas)
- Chunk: 10 (mejor parámetro según la pruebas realizadas)
- Cores: 1 (se utiliza 1 core por cada proceso utiliza su propia caché. Por lo tanto, para distinguir mejor el impacto de la caché es utilizar sólo un core)
- Memoria usada por el programa: 576 MB (total de espacio que utiliza esta ejecución según los cálculos teóricos).
- Pool : 0 (no procede usar el pool porque rellenaría la caché en adelanto y si no cupieran los datos debería hacer las peticiones dos o más veces).
- Política reemplazo: LRU (mejor parámetro según la pruebas realizadas).

100% (>576MB)	75% (432MB)	50% (288MB)	25% (144MB)	10% (58MB)	5% (29MB)	1% (6MB)
15’38’’	16’54	22’04’’	37’00’’	104’57’’	369’14’’	1878’37’’

Si observamos los resultados podemos ver que el uso de la caché aumenta mucho el rendimiento. Cuando la caché permite almacenar todos los datos (100%) de todas la peticiones (por lo tanto se hacen el menor número de peticiones a Cassandra) se obtiene el mejor tiempo. Se puede observar que a medida que el tamaño se va reduciendo el tiempo empeora hasta llegar a tiempos muy elevados cuando prácticamente no hay caché (1%).

6.5. Pool

El propósito de las siguientes ejecuciones es ver cuál es la mejora del rendimiento usando el pool.

A continuación se presenta la comparativa de tiempos obtenidos para una misma configuración y distintos tamaños de pool.

Parámetros por defecto:

- Secuencia: rrm_1000 (en este caso se aumenta el dataset de 500 a 1000 secuencias para encontrar un mayor impacto del uso del pool).
- Nodos C*: 4 (mejor configuración según pruebas anteriores).
- Chunk: 10 (mejor configuración según pruebas anteriores).
- Memoria máxima: 7500 MB (No procede porque cabe en memoria).
- Cores: 4 (mejor configuración según pruebas anteriores).
- Política Reemplazo : LRU (No procede porque cabe en memoria).

Pool 0	Pool 10
57’59’’	**31’42’’**

Si observamos los resultados vemos que con el uso del pool se obtiene mejor tiempo. Este hecho es debido a que el pool se baja la librería en adelanto para cargarla en la caché mientras la aplicación realiza otras tareas. Además, lo hace de forma concurrente que implica que para un pool de 10 threads, cada thread haga peticiones a Cassandra a la vez y rellenen la caché a la vez.

6.6. Resultados generales.

Las siguiente ejecuciones pretenden mostrar los tiempos obtenidos para la ejecución de distintos conjuntos de secuencias para SCoffee y T_Coffee.

Se presentan las tablas comparativas y distintos gráficos de los tiempos obtenidos para una misma configuración y distintos datasets.

Parámetros por defecto (se han elegido los óptimos según los resultados anteriores):

- Nodos C*: 4
- Chunk: 10
- Memoria máxima: 7500/5500/4000 MB (7500MB para todas excepto las de 2000; 5500MB para todas las de 2000 excepto las mayores; 4000MB para las 2 mayores Acetyl_2000 y sdr_2000)
- Cores: 4/3 (3 para la Acetyltransf_2000)
- Pool: 10 (ó 0 para las secuencias de 2000v ya que no caben en memoria)
- Política Reemplazo : LRU
- CPU’s Spark: 60

Dataset	Tiempo Spark	Tiempo Alineamiento	Tiempo total SCoffee	Tlempo total T_Coffee
Acetyltransf_100	56’’	59’’	1’55’’	**1’22’’**
Acetyltransf_200	2’ 45’’	3’03’	5’48’’	**5’44’’**
Acetyltransf_500	14’50’’	18’34’’	**33’24’’**	43’04’’
Acetyltransf_1000	78’49’’	83’10’’	**161’59’’**	No cabe
Acetyltransf_2000	231’58’’	1045’59’’	**1277’57’’**	No cabe

rrm_100	1’02’’	28’’	1’30’’	**51’**
rrm_200	2’50’’	1’25’’	4’15’’	**3’09’’**
rrm_500	10’25’’	7’43’’	**18’08’’**	23’14’’
rrm_1000	43’01’’	31’42’’	**74’43’’**	138’24’’
rrm_2000	168’11’’	162’04’’	**330’15’’**	No cabe
rvp_100	45’’	13’’	58’’	**34’’**
rvp_200	2’05’’	51’’	2’56’’	**2’14’’**
rvp_500	11’05’’	5’18’’	**16’23’’**	17’25’’
rvp_1000	46’41’’	21’39’’	**68’20’’**	96’26’’
rvp_2000	187’55’’	162’04’’	**349’59’’**	No cabe
zf-CCHH_100	43’’	7’	50’’	**5’’**
zf-CCHH_200	2’02’’	10’’	2’12’’	**21’’**
zf-CCHH_500	10’20’’	1’01’’	11’21’’	**5’36’’**
zf-CCHH_1000	41’30’’	4’54’’	46’24’’	**39’47’’**
zf-CCHH_2000	174’02’’	29’49’’	**203’51’’**	No cabe
sdr_100	1’20’’	2’05’’	**3’25’’**	4’33’’
sdr_200	3’40’’	8’37’’	**12’17’’**	16’14’’
sdr_500	19’11’’	40’56’’	**60’07’’**	106’46
sdr_1000	72’42’’	188’51’’	**261’33’’**	No cabe
sdr_2000	-	-	-	No cabe

A continuación se pueden ver los mismos resultados finales en formato gráfico.

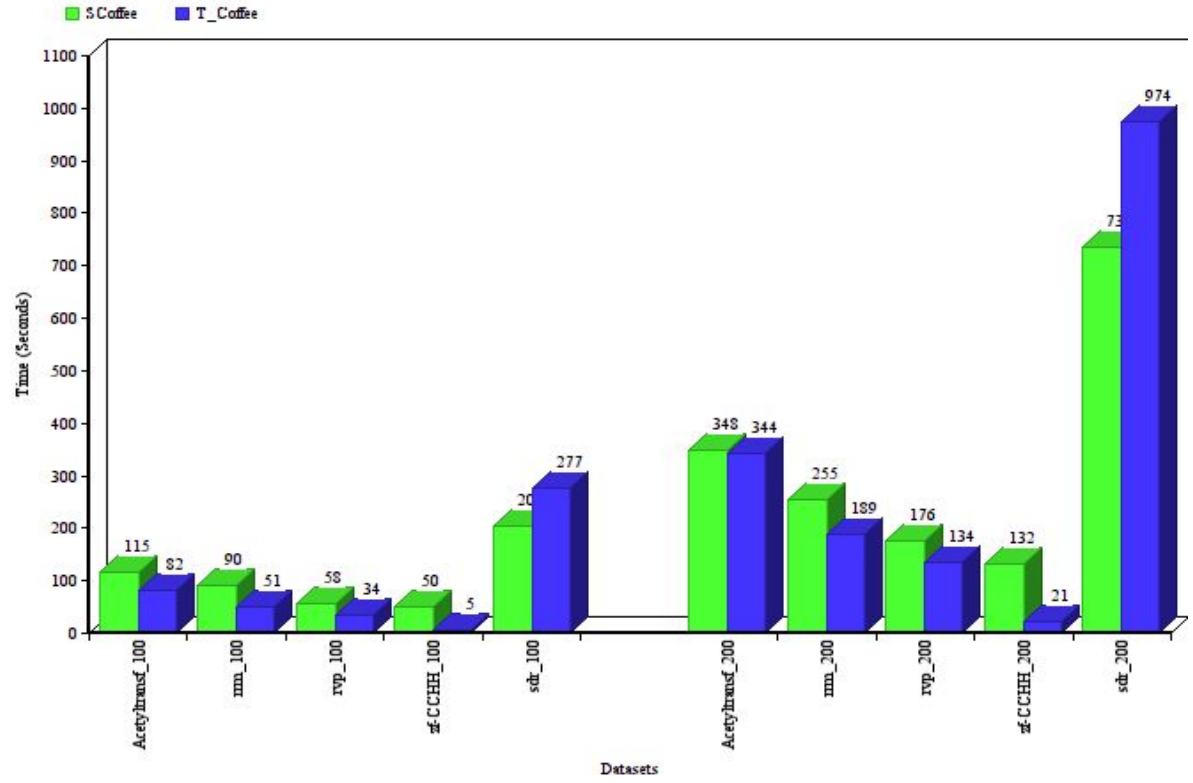

Figura 8: Tiempos de SCoffee y T_Coffee para datasets de 100 y 200 secuencias

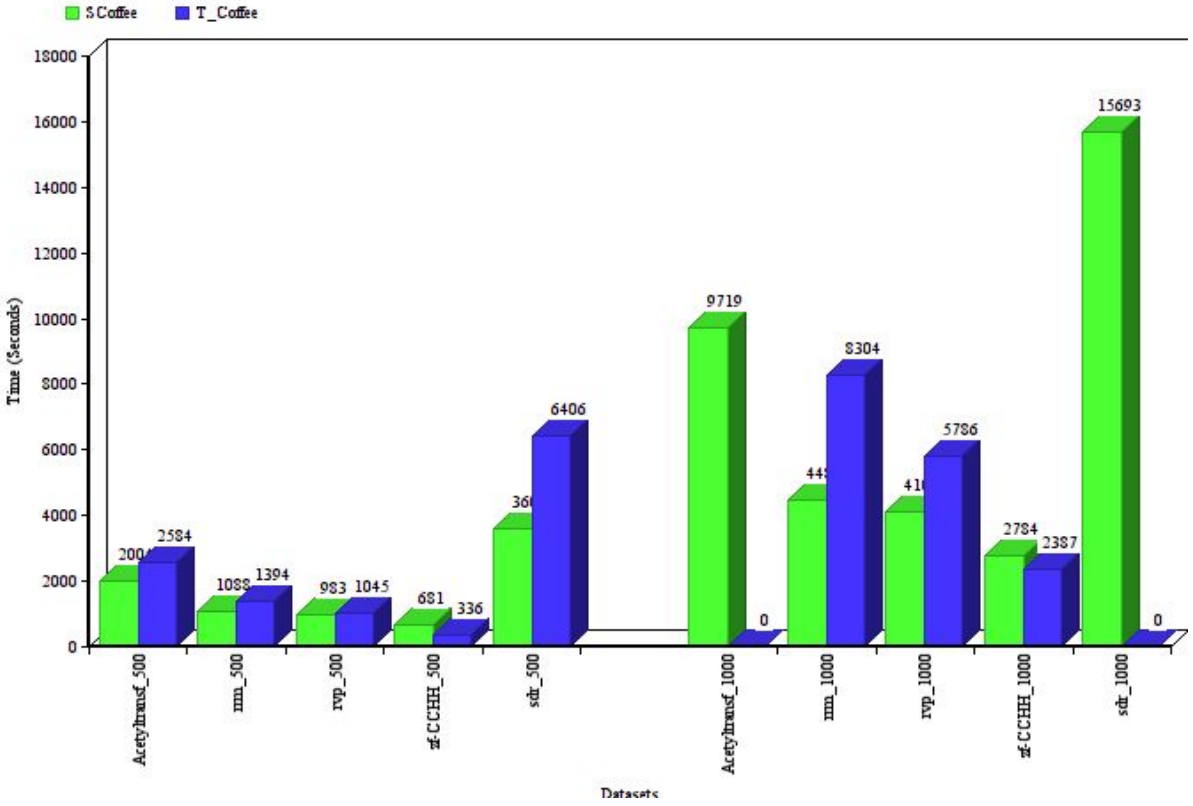

Figura 9: Tiempos SCoffee y T_Coffee para datasets de 500 y 1000 secuencias

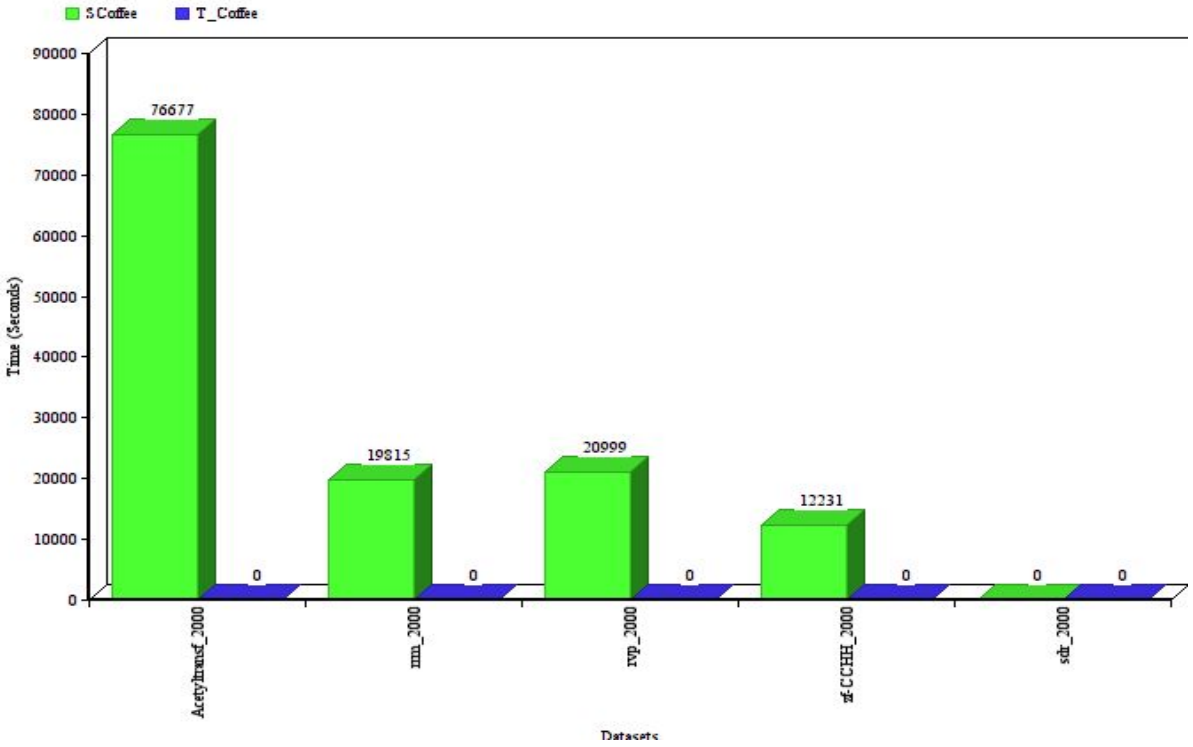

Figura 10: Tiempos SCoffee para datasets de 2000 secuencias

Si observamos los resultados y las gráficas vemos que SCoffee ha permitido alinear datasets de muchas secuencias (algunos de 1000 y algunos de 2000) mientras que T_Coffee no. También se observa que para datasets de secuencias largas (sdr, por ejemplo) o datasets de muchas secuencias SCoffee calcula la librería y alínea en menor tiempo que T_Coffee.

Con respecto al alineamiento obtenido se puede observar que no se pierde calidad con respecto a T_Coffee, ya que una vez ejecutado el Balibase (Benchmark Alignment Database) con T_Coffee y SCoffee se obtienen las siguientes medias (para el sum-of-pairs -SP-) de los datasets que lo componen.

	RV11	RV12	RV20	RV30	RV40	RV50	Total SP
T_Coffee	0.534	0.879	0.827	0.718	0.758	0.759	0.743
SCoffee	0.590	0.871	0.868	0.757	0.748	0.772	0.768

7.CONCLUSIONES.

Como conclusiones locales (a nivel de parámetros) encontramos que como más nodos se añadan a Cassandra más rápida es la ejecución. Esto implica que a más nodos mayor rendimiento, ya que Cassandra escala linealmente.

Para al chunk se observa que el mejor tiempo obtenido es para el chunk 10. Para este parámetro nos hemos de basar en el método empírico ya que no se observa ninguna relación entre diferentes chunks.

Para la política de reemplazo observamos que se obtiene mejor resultado en la LRU que en la Random. Este resultado se ciñe a lo esperado. Este resultado es debido a la forma en que T_Coffee construye el alineamiento (llamando reiteradas veces por el mismo par secuencia-residuo) que implica que la LRU optimice el tiempo empleado mejor que cualquier otra política.

Para el uso de memoria se observa el impacto de la caché. Se puede observar que cuando prácticamente no hay caché (1% de memoria) el tiempo empleado es mucho mayor que cuando están todos los datos de la caché (100%). Por lo tanto con el uso de esta estructura de datos se mejora mucho el rendimiento de la aplicación.

Para el pool se observa que es mejor usar el pool ya que éste puede hacer peticiones a Cassandra concurrentemente y descargar la librería en adelanto mientras T_Coffee hace otras operaciones. Con esta herramienta se mejora el tiempo de alineamiento siempre y cuando quepa la librería en la caché ya que sino durante las peticiones concurrentes llegaría un momento que no cabría en memoria y debería hacer vaciado de caché. Este hecho implicaría hacer algunas peticiones, al menos, dos veces (una para el pool y si se libera otra cuando hace el alineamiento).

A nivel de conclusiones globales se observa que se han cumplido los objetivos del proyecto ya que, tal como se pretendía, se ha mejorado la escalabilidad de T_Coffee.

Si se hace un estudio más detallado de los resultados se observa que T_Coffee és más rápido que SCoffee para datasets de pocas secuencias o de secuencias de poca longitud. Para datasets grandes o de secuencias de mucha longitud es mejor SCoffee. Este hecho es debido a que mientras TCoffee calcula la librería en una sola computadora, SCoffee permite calcularla con tantos cores como haya disponibles para Spark.

Para el alineamiento, T_Coffee siempre será más rápido que SCoffee debido a que consultar a memoria es más rápido que hacer peticiones a una base de datos (Cassandra en este caso). El uso de la caché permite aumentar mucho el rendimiento (sobretodo cuando T_Coffee necesita utilizar la Swap cuando consume la RAM, que es más lenta que hacer peticiones a Cassandra) que implica en general que para algunos datasets no muy grandes (500 secuencias) se obtengan mejores tiempos en SCoffee.

Finalmente, concluir que SCoffee ha permitido alinear datasets de 2000 secuencias (y algunos de 1000) que T_Coffee no permite debido a que no caben en memoria.

8. FUTURAS IMPLEMENTACIONES.

Como línea abierta queda el implementar el pool de threads de manera que vaya siguiendo el árbol guía a la hora de lanzar peticiones a Cassandra y rellenar la caché de consistencia. De esta manera el pool rellenará la caché empezando por los datos relacionados a lo pares secuencia-residuo que primero hagan falta.

9. BIBLIOGRAFÍA Y WEBGRAFÍA.

- A foreign function library for Python [Data consulta 25/11/16] https://docs.python.org/2/library/ctypes.html
- Apache Spark Tutorial [Data consulta 14/12/2016] https://www.tutorialspoint.com/apache_spark/
- Datastax C/C++ Driver for Apache Cassandra [Data consulta 12/12/2016] https://github.com/datastax/cpp-driver
- Hadoop Tutorial [Data consulta 22/12/2016] https://www.tutorialspoint.com/es/hadoop/
- Introduction to Cassandra Query Language [Data consulta 08/01/2017] https://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html
- IPC:Shared Memory [Data consulta 25/03/2017] https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html
- Modern and efficient C++ Thread Pool Library [Data consulta 08/01/2017] https://github.com/vi-vit/ctpl
- Standard C++ Library Reference [Data consulta 11/11/2016] http://www.cplusplus.com/reference/
- T_Coffee Manual [Data consulta 12/11/2016] https://media.readthedocs.org/pdf/tcoffee/latest/tcoffee.pdf
- T_Coffee Multiple Sequence Alignment Tools [Data consulta 14/11/2016] http://www.tcoffee.org/Projects/tcoffee/