UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Grid y Computación de Altas Prestaciones
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Paralelización de biomarcadores de perfusión a través de un modelo de cómputo en la nube

Trabajo Fin de Master

Master en Computación Paralela y Distribuida

Autor: Gil Arcas, Enrique

Tutor: Segrelles Quilis, José Damián

Cotutor: Blanquer Espert, Ignacio

2015-16

Resumen

Actualmente, en el hospital Universitario y Politécnico de la Fe (HPULF) se utilizan en la práctica clínica biomarcadores basados en Imagen médica como herramienta de apoyo en el diagnóstico de diferentes enfermedades. Uno de estos biomarcadores se basa en el análisis de la perfusión (captación de contraste en el tejido) mediante el modelado de curvas de señal-tiempo relativa a la absorción de un contraste por parte de los tejidos. Esto permite evaluar la permeabilidad tisular, basándose en la forma de las curvas y el lavado del medio de contraste. Este biomarcador permite evaluar la angiogénesis asociada a tejidos tumorales, los cuales presentan una serie de características vasculares que los distinguen de los tejidos sanos del organismo. El cómputo de dicho biomarcador tiene en la actualidad un coste del orden de magnitud de horas y está implementado en Matlab. El trabajo a realizar en este TFM ha consistido en el estudio y análisis de dicho biomarcador y en la propuesta de diferentes enfoques que exploten las características de elasticidad de la computación en la nube, con el objeto de rebajar el coste temporal utilizando recursos computacionales de nube pública, realizando una inversión mínima para su puesta en marcha. Los resultados de este trabajo, serán las implementaciones correspondientes y un estudio del coste temporal a través de un conjunto de casos reales.

Palabras clave: Cloud, Biomarcador, Perfusión

Abstract

Nowadays, biomarkers based on medical resonance imaging (MRI) are used in the Hospital Universitario y Politécnico de la Fe (HPULF) to support the diagnosis of different diseases. One of these biomarkers is focused on the perfusion analysis modelling signal-time curves relative to contrast absorption. This biomarker allows evaluating the angiogenesis associated to tumour tissues. Currently, the computation time of this biomarker has a magnitude order of dozens of hours. The aim of this paper is to show the benefits of using the Cloud architecture to this kind of algorithms in order to reduce de execution time.

Tabla de contenidos

1. Introducción. 6
2. Materiales y métodos. 9
  2.1. Biomarcador de Perfusión basado en RM 9
  2.2. Requisitos para la optimización 11
  2.3. Reimplementación Secuencial Optimizada. 12
  2.4. Reimplementación Paralela Optimizada 12
  2.5. Requisitos de la solución Cloud 14
    2.5.1. Arquitectura cliente-servidor. 14
    2.5.2. Conexiones HTTP. 15
    2.5.3. Cálculo de forma síncrona. 16
    2.5.4. Granularidad. 16
    2.5.5. Tolerancia a fallos. 16
    2.5.6. Proveedores Cloud 17
  2.6. Datasets e Infraestructura 19
3. Resultados y Discusión. 21
  3.1. Algoritmos reimplementados 21
  3.2. Arquitectura de la solución Cloud 21
  3.3. Estudio y Análisis Comparativo 22
  3.4. Reimplementación Secuencial Optimizada. 22
  3.5. Reimplementación paralela optimizada. 23
  3.6. Solución Cloud. 23
  3.6.1. Ratio clientes / servidores. 23
  3.6.2. Test de escalabilidad 25
  3.6.3. Algoritmo original vs solución Cloud optimizada 27
  3.6.4. Elasticidad. 28
4. Conclusiones 32
5. Referencias 33
ANEXO 1 37
  PLANIFICACIÓN Y EJECUCIÓN DE TAREAS 37

Introducción.

Hoy en día la comunidad científica es muy prolífica, consiguiendo nuevos e importantes avances en todas sus disciplinas. Una de los principales causas que ha propiciado estos avances, ha sido la aplicación y uso de la computación científica en los proyectos de investigación [00] [01] [02].

En el área de la salud, los campos relacionados con el diseño e implementación de biomarcadores de Imagen Médica han sido de los más beneficiadas, facilitando el diagnóstico o seguimiento de muchas enfermedades que prevalecen en la sociedad actual, como por ejemplo las enfermedades oncológicas. Los departamentos de radiología se han nutrido de herramientas de proceso muy potentes de imagen [03], que actúan como biomarcadores y se han integrado en las Workstation de los Picture Archiving and Communication System (PACS) [04] de los hospitales, permitiendo que muchos pacientes en los primeros estadios del cáncer hayan sido diagnosticados y por lo tanto hayan aumentado sus posibilidades de supervivencia. Además del diagnóstico precoz, estos nuevos biomarcadores han permitido la estratificación de la gravedad de la enfermedad, la planificación de sus tratamientos y la evaluación de la efectividad de los mismos.

En la actualidad, para desarrollar este tipo de herramientas, es muy común que un investigador utilice algún *framework* de cómputo científico para poder modelar e implementar prototipos o simulaciones que le ayuden a corroborar o descartar sus biomarcadores [05]. Uno de los *frameworks* más extendidos es Matlab [06], propiedad de MathWorks Inc. El uso de esta herramienta, es muy común en el área de la Imagen Médica, como podemos ver en [07], en el que se desarrolla un prototipo para la detección del disco ocular en imágenes de retinas, o en [08] en el que se desarrolla otro prototipo para la detección y extracción de tumores cerebrales basándose en imágenes.

La clave del éxito de Matlab entre la comunidad científica es su gran versatilidad y productividad en el desarrollo de prototipos, lo cual permite implementar y validar los modelos que definen los biomarcadores de forma rápida, dado que pone a disposición de sus usuarios una completa gama de funciones para el análisis y procesamiento de diferentes tipos de datos como por ejemplo las imágenes médicas. Por el contrario, uno de los principales problemas que presenta es su ineficiencia de cómputo a la hora de ejecutar los modelos que implementa. A pesar de ello, ocurre con frecuencia que los propios prototipos generados y validados a través de Matlab sean utilizados directamente en la práctica clínica, aun siendo ineficientes, por lo que en algunas ocasiones es prácticamente imposible ponerlos en marcha en la práctica clínica diaria dada su ineficiencia y coste temporal, relegando a estos a un uso limitado o incluso a no utilizarlo. Este hecho es debido a que en muchos casos, para hacer estos modelos más eficientes, la

reimplementación de los modelos a través de otros frameworks basados en lenguajes más óptimos (como los basados en lenguajes como C++) no es suficiente, y se requiere del conocimiento de modelos de programación complejos y muy específicos como los basados en la computación paralela [09][10], computación Grid [11] o computación en la nube [12], capaces de escalar los recursos de computo con el objeto de optimizar la ejecución de los algoritmos hasta el punto de que su coste temporal sea viable para su aplicación en la práctica clínica diaria. Por desgracia, estos modelos de programación requieren de un experto informático formado en dichas disciplinas, del que por lo general no se dispone en los hospitales y en muchos casos, el coste de los recursos de cómputo requeridos puede ser inabordable para los centros médicos. Además, las soluciones que surgen de estos modelos requieren de nuevo equipamiento, que no siempre son sencillos de integrar en los sistemas hospitalarios, dado que están sometidos a importantes restricciones para preservar la seguridad de sus sistemas.

El contexto en el que se enmarca este trabajo se sitúa en el hospital Universitario y Politécnico de la Fe (HPULF), en el cual se utilizan en la práctica clínica biomarcadores basados en Imagen médica para apoyar en el diagnóstico de diferentes enfermedades. Uno de estos biomarcadores, desarrollado por la empresa valenciana QUIBIM S.L.[13], se basa en el análisis de la perfusión (captación de contraste en el tejido) mediante el modelado de curvas de señal-tiempo relativa a la absorción del contraste [14]. Esto permite evaluar la permeabilidad tisular, basándose en la forma de las curvas y el lavado del medio de contraste además de evaluar la angiogénesis asociada a tejidos tumorales, los cuales presentan una serie de características vasculares que los distinguen de los tejidos sanos del organismo. Este biomarcador ya ha sido validado [15], sin embargo, su implementación es la heredada del prototipo desarrollado en Matlab y tiene un coste del orden de magnitud de horas, por lo que su uso en la práctica clínica está muy limitado.

En los últimos años estamos viviendo una revolución tecnológica a nivel mundial con la aparición de las plataformas Cloud. Estas plataformas permiten al usuario (empresas que quieren dar un servicio WEB) el aprovisionarse de recursos de manera dinámica (en función de la carga de trabajo) pagando solo por su uso (horas consumidas de recursos) sin necesidad de realizar un desembolso inicial para el aprovisionamiento de equipos. Con ello, una pequeña empresa puede escalar su servicio de una manera muy sencilla lo que hasta ahora era impensable. Por todo esto, QUIBIM S.L quería estudiar la posibilidad de ofrecer el biomarcador para el análisis de la perfusión como un *Software as a Service* (SaaS), con la idea de crear el programa con una estructura cliente-servidor que se pudiera desplegar en un entorno *Cloud*, lo que permitiría dar el servicio a una escala global aprovechando las propiedades de escalabilidad y elasticidad inherentes a las arquitecturas en la Nube y a un coste asumible por los centros que lo utilizasen.

Por ello, el objetivo de este TFM ha sido diseñar una solución en la nube con el objeto de mejorar los tiempos de ejecución del biomarcador de perfusión y de esa forma que sea viable su uso en la práctica clínica diaria y que a su vez, dicha solución sea fácil de integrar en los sistemas hospitalarios y con un coste asumible por el centro médico que lo utilice. Para la consecución de dicho objetivo, se plantea el estudio del biomarcador y su optimización mediante su reimplementación en un lenguaje más eficiente y aplicando conceptos de computación paralela (aprovechando que hoy en día los equipos informáticos disponen de multiprocesadores) y diseñando e implementando una solución que permita la computación en la nube del biomarcador. Además, como objetivo específico se plantea un análisis comparativo de la eficiencia de la nueva solución respecto a la solución inicial del biomarcador mediante casos reales proporcionados por el HUPLF.

2. Materiales y métodos.

En esta sección, en primer lugar, se describe en términos generales el algoritmo de perfusión y algunos detalles técnicos del algoritmo original implementado en Matlab por QUIBIM S.L. A continuación, se describe los aspectos técnicos relativos al lenguaje de programación, librerías y modelo de programación de computación paralela utilizados para la reimplementación del biomarcador, así como una justificación de la elección de los mismos entre las diferentes alternativas existentes. En la siguiente subsección se detallan los requisitos de la solución en la nube planteada como objetivo, así como un estudio de aquellos componentes existentes que pueden implementar dichos requisitos y la justificación de los componentes seleccionados para diseñar la arquitectura. Finalmente, se describen los *datasets* empleados para el estudio comparativo realizado, facilitado por el HUPLF.

2.1. Biomarcador de Perfusión basado en RM

El biomarcador de Perfusión se basa principalmente en la medición de la capacidad de absorción de un contraste por parte de los tejidos. Los tejidos cartilaginosos normales y los tumorales tienen una permeabilidad capilar y un volumen de intercambio intersticial distintos que pueden ser detectados por resonancia magnética mediante la aplicación de un contraste en el sistema vascular y la posterior medición en distintos intervalos de tiempo de las fluctuaciones del nivel de acumulación de dicho contraste en dichos tejidos. En la 0 se muestra la fórmula que gobierna la evolución de la concentración de contraste en el espacio extracelular extravascular externo:

$$C_t(t) = v_p C_p(t) + K^{\text{trans}} \int_0^t C_p(u)\, e^{-k_{ep}(t-u)}\, du$$

Figura 1: Función para el cálculo de la evolución del contraste.

Para cualquier duda sobre los parámetros de la anterior función puede consultar la referencia número [13]. En la figura 2 podemos ver una representación del proceso de perfusión. El modelo representado por la fórmula mostrada en la 0, permite inferir, a partir de las mediciones realizadas por resonancia magnética, tres marcadores que son:

- K^{trans}: Permeabilidad entre venas y la zona extracelular externa.
- Ve: Volumen del espacio extracelular extravascular externo.
- K_{ep}: Ratio de extracción entre la zona extracelular externa y las venas.

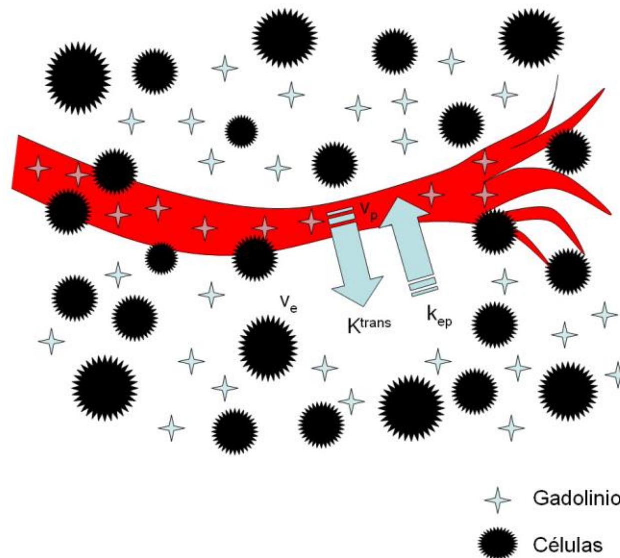

Figura 2: Representación gráfica del proceso de perfusión en el tejido intersticial .

Los valores de estos tres marcadores sufren incrementos conforme la enfermedad avanza en el paciente, por lo que el poder realizar estas mediciones por resonancia magnética facilita enormemente su seguimiento y medir el grado de agresividad de los tumores.

El algoritmo que implementa el modelo formulado en la 0, está implementado a través de un script de Matlab. Como datos de entrada recibe una matriz de cuatro dimensiones a la que llamamos "volumen", correspondientes a todas las imágenes que conforman los cortes adquiridos en el estudio de resonancia Magnética. Las tres primeras dimensiones del volumen marcan el valor de las coordenadas geométricas de cada voxel[1] en [1]los ejes X, Y y Z. La cuarta dimensión marca los instantes temporales en los que se ha tomado cada medición. El volumen se carga desde un archivo con formato NIFTI [16] que contiene además metadatos asociados al mismo, como nombre del paciente, modalidad (en este caso resonancia magnética), parámetros utilizados por el dispositivo de captación del estudio, entre otros….

Posteriormente a la carga y para realizar la inferencia de los tres marcadores se ejecuta un ajuste de mínimos cuadrados de las mediciones tomadas en cada voxel independientemente del resto de voxeles. La función de Matlab que realiza este procedimiento se llama *lsqcurvefit* [17]. *Lsqcurvefit* precisa como parámetro de una función modelo que determina como se realiza el ajuste. Una vez se ha iterado sobre todos los voxeles del volumen y se ha realizado el ajuste se

[1] Vóxel: El vóxel es la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, el equivalente del píxel en un objeto 3D.

devuelven al usuario tres matrices tridimensionales que contienen respectivamente los valores de los marcadores k^{trans}, v_e y k_{ep} para cada voxel.

```
1   #Carga de datos
2   volumen = carga_volumen_desde_archivo()
3   func_modelo = carga_modelo_desde_archivo()
4   parametros = carga_parametros_desde_archivo()
5   #Variables
6   d1 = length(volumen[X])
7   d2 = length(volumen[Y])
8   d3 = length(volumen[Z])
9   d7 = length(volumen[T])
10  #Matrices de salida
11  ktrans_matrix = array[d1, d2, d3]
12  kep_matrix = array[d1, d2, d3]
13  ve_matrix = array[d1, d2, d3]
14  #Bucles de proceso
15  for x in d1:
16    for y in d2:
17      for z in d3:
18        voxel_measurements = volume[x,y,z]
19        voxel_ktrans, voxel_kep, voxel_ve = lsqcurvefit(voxel_measurements, model_func)
20        ktrans_matrix[x, y, z] = voxel_ktrans
21        kep_matrix [x, y, z] = voxel_kep
22        ve_matrix [x, y, z] = voxel_ve
23  #Devolución de resultados
24  return ktrans_matrix, kep_matrix, ve_matrix
```

Figura 3: pseudocódigo del algoritmo original

Como podemos ver en la figura 3, el programa original está constituido como un monolito de procesamiento serial en el que se distinguen tres bloques: el de carga del volumen a procesar, el de procesamiento que itera sobre todos los voxeles y los procesa y el que devuelve el resultado al usuario.

2.2. Requisitos para la optimización

Durante la reunión para fijar objetivos que aparece en el anexo de planificación se acordaron una serie de requisitos que debía cumplir el algoritmo resultado de este proyecto:

R1: Se debía reimplementar el código a un lenguaje de programación más apropiado para computo de altas prestaciones.

R2: Se había de mantener la interfaz de usuario (IU) Matlab del biomarcador existente para la carga de datos y recuperación de resultados, el motivo de esta decisión fue que no se deseaba cambiar la interacción con los usuarios finales del biomarcador, ya que estos módulos no suponían una carga de computo significativa y de esa manera se conseguía que los cambios en el módulo de proceso fueran transparentes al usuario, facilitando así su integración en el flujo de trabajo del hospital.

R3: Estudiar la posibilidad de implementar el algoritmo con una arquitectura Cloud para tener la posibilidad de ofrecerlo como SaS.

2.3. Reimplementación Secuencial Optimizada.

El foco de este TFM se centra en la optimización del biomarcador presentado en la sección 2.1 por tanto cumpliendo con los requisitos acordados en 2.2.. La primera tarea en el TFM fue la de reimplementar el módulo de procesamiento en un lenguaje más eficiente y permitir su integración en los IU Matlab Existentes. La decisión fue sencilla, dado que el propio Matlab ofrece los ficheros MEX [18] los cuales permiten hacer de puente entre Matlab y los lenguajes de programación C, C++ y Fortran. Los ficheros MEX no son más que un trozo de código escrito en C, C++ o Fortran que expone una función con una determinada interfaz que permite ser llamada desde Matlab, dentro de dicha función se puede insertar toda la funcionalidad que queramos utilizando cualquiera de esos tres lenguajes. Dado que hoy día Fortran tiene un uso casi residual comparado con C y C++, y que C++ enriquece C con mucha funcionalidad, como por ejemplo el uso de *templates* o la programación orientada a objetos, elegimos C++ como lenguaje para reimplementar el bloque de procesado. Además, bien es conocido la eficiencia de este lenguaje y su uso para el desarrollo de muchas librerías de referencia que implementan funciones de procesamiento de imagen como son DCMTK [19], VTK [20] o ITK [21].

Seleccionado C++ como lenguaje de programación para la reimplementación, quedaba encontrar una librería en C++ que nos permitiera realizar el ajuste de curvas por mínimos cuadrados, tal y como se ha comentado en la sección 2.1. Se estudiaron algunas librerías *open source,* como por ejemplo MPFIT [22] o GNUGSL [23] pero nos decidimos por Ceres Solver [24] la cual permite resolver ajustes no lineales de mínimos cuadrados con límites y tiene una gran aceptación en la comunidad científica. Esta librería requiere que el ‘modelo’ que se le pasa como parámetro a la función que realiza el ajuste esté implementado como un *template*, lo que permite al compilador hacer *unrolling* del código, generando unos ejecutables mucho más eficientes que en una versión sin *templates*.

Los resultados de la optimización comparados con el algoritmo Matlab son presentados en los apartados 3.3 y 3.4. A pesar de que fueron satisfactorios, estos fueron insuficientes para su puesta en marcha en la práctica clínica diaria, por ello se decidió optimizar esta nueva versión utilizando modelos de programación basados en la computación paralela que permitieran aprovechar los recursos existentes al máximo sin requerir de ninguna inversión adicional.

2.4. Reimplementación Paralela Optimizada

La solución presentada en el apartado 2.3 solo estaba aprovechando uno de los *cores* disponibles en la CPU del recurso donde se ejecuta. En la actualidad, los procesadores son

prácticamente en su totalidad *multicore* (Intel i3, Intel i5, Inter I7, AMF FX…) y por tanto se podría aprovechar esta característica para desarrollar una nueva implementación paralelizada más óptima que utilizara el máximo número de *cores* disponibles.

La sección de código susceptible de mejora por técnicas de paralelización era la correspondiente a las líneas 15 a la 22 del pseudocódigo mostrado en la figura 3. Solo nos faltaba un detalle, elegir la granularidad adecuada para realizar la paralelización. Puesto que los volúmenes de datos que nos facilitaron corresponden a una representación tridimensional de órganos humanos, decidimos dividir el volumen a través del eje Z como si hiciéramos cortes horizontales de los órganos [figura 4], haciendo que en cada iteración se procesara un corte.

Figura 4: Cortes horizontales de un cerebro que simulan nuestra unidad de granularidad

Como en los volúmenes el tamaño del eje Z oscilaba entre 100 y 200 cortes obtuvimos un particionado de esa magnitud. Cabe tener en cuenta que cada voxel puede tener un tiempo de procesamiento distinto, ya que depende del proceso de ajuste de mínimos cuadrados, lo que podría desembocar en un reparto irregular de la carga entre los procesadores; pero esto no sucede gracias al reparto de la carga dinámico realizado y a la óptima granularidad elegida.

Existen varias tecnologías para realizar una paralelización de código a nivel local, la más extendida es OpenMP [25] pero también se puede usar MPI [26] o la librería Pthreads [27]. OpenMP es una librería de alto nivel que utiliza un modelo de memoria compartida y mediante la inserción de directivas en el código se puede gestionar el uso de hilos de ejecución. La librería MPI está pensada para funcionar con un modelo de memoria distribuida entre diferentes CPUs, pero en las últimas versiones se han implementado mejoras para la gestión de los procesos residentes en una misma CPU no teniendo nada que envidiar en rendimiento a OpenMP. Por

último, la librería Pthreads, es la más antigua de las tres, es de un nivel más bajo por lo que requiere del programador mucho más esfuerzo para implementar la gestión de los hilos lo que hace que actualmente este en desuso en favor de OpenMp y MPI.

Finalmente se decidió optar por la solución ofrecida por Matlab a través del módulo MatlabPool, lo cual permite la gestión de *threads* dentro de Matlab. Dicho modulo incluye la primitiva *parfor* que es muy similar a '*pragma omp parallel for*' de la librería OpenMP. Con MatlabPool puedes crear un conjunto de *threads* o *pool* y después, al construir un bucle con *parfor* [figura 5], se irán asignando iteraciones a los *threads* que estén disponibles de manera dinámica, lo que favorece un mejor reparto de la carga que si hubiéramos repartido las iteraciones de forma lineal sobre los *threads*.

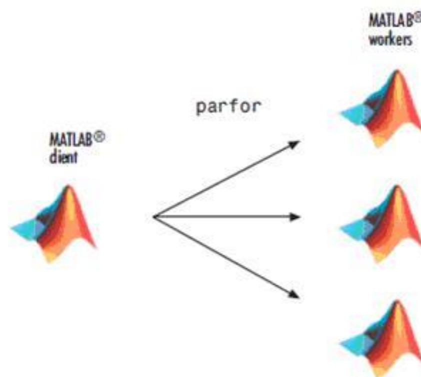

Figura 5: Representación de un pool de workers o Threads en Matlab .

2.5. Requisitos de la solución Cloud

Para desarrollar la solución Cloud [28] propuesta como objetivo de este TFM, se añadieron un conjunto de requisitos secundarios para poder adaptar el algoritmo a una arquitectura en la nube, y que se detallan en la siguientes subsecciones:

2.5.1. Arquitectura cliente-servidor.

Este modelo es la base de todo programa desplegado en el Cloud. De un lado están los Clientes, los cuales se encargan únicamente de enviar peticiones de trabajo al Cloud y de recibir los resultados al finalizar. En el otro lado están los servidores, que son los que realizan las operaciones de cómputo para servir los resultados de las peticiones que les llegan a través de Internet [figura 6].

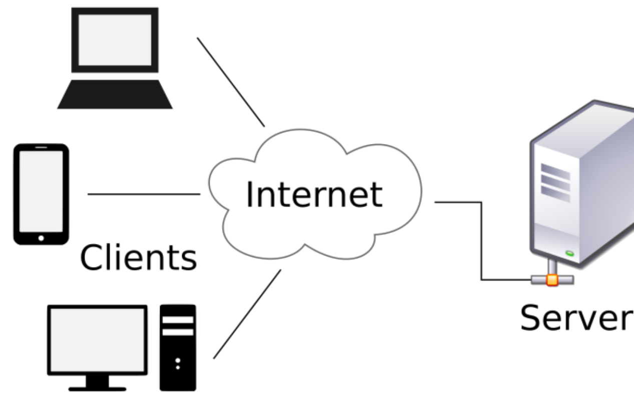

Figura 6: Arquitectura cliente-servidor.

En el lado del cliente tenemos Matlab que carga el volumen de datos, genera un *pool* de *threads* y particiona el volumen mediante un bucle *parfor* asignando dinámicamente iteraciones a los *threads* libres. Cuando un *thread* recibe un conjunto de datos los envía al entorno C++ mediante un archivo MEX, ya en C++ hemos utilizado la librería *libcurl* [29] para realizar una petición de computo *post* al servidor enviándole los datos. Cuando el servidor termina devuelve la respuesta y se le pasa al cliente Matlab.

En la parte del servidor tenemos un servicio HTTP implementado totalmente en C++ con ayuda de la librería *libmicrohttpd* [30]. Está escuchando peticiones de trabajo a través de Internet en un puerto y cuando le llega una utiliza Ceres Solver para realizar el cómputo y devolver una respuesta.

Con esta arquitectura ganamos mucha versatilidad, el requisito para que un usuario pueda procesar un volumen es tener un equipo con Matlab, no hace falta que el equipo sea especialmente potente ya que el computo se va a realizar en un servidor remoto que puede ofrecer servicio a múltiples clientes.

2.5.2. Conexiones HTTP.

Elegimos el estándar HTTP [31] como protocolo de comunicación entre clientes y servidores porque es muy sencillo a la par que eficiente y eficaz. "Hypertext Transfer Protocol o HTTP (en español protocolo de transferencia de hipertexto) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. HTTP fue desarrollado por el World Wide Web Consortium y la Internet Engineering Task Force, colaboración que culminó en 1999 con la publicación de una serie de RFC, el más importante de ellos es el RFC 2616 que especifica la versión 1.1. HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, *proxies*) para comunicarse. HTTP es un

protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores" (Wikipedia).

HTTP describe un conjunto de métodos de aplicación que se utilizan para distinguir entre diferentes tipos de comunicaciones. En nuestro caso nosotros vamos a enviar desde el cliente un conjunto de datos relativamente grande al servidor que lo procesará y devolverá los resultados. Este patrón encaja perfectamente con el método de aplicación *post*.

2.5.3. Cálculo de forma síncrona.

Es muy típico, en este tipo de algoritmos de procesamiento de grandes volúmenes de datos, el realizar el cálculo de una forma asíncrona, es decir, enviando el volumen de datos al servidor que nos devolvería un identificador mediante el cual podemos ir preguntando con posterioridad por el estado del trabajo y recoger los resultados una vez finalizado. Pero dado que nuestras pruebas con las mejoras anteriores nos dieron tiempos de ejecución suficientemente cortos, decidimos realizar el procesamiento de una manera síncrona, lo que nos simplifica mucho la gestión de las conexiones y el mantener servidores libres de estado.

2.5.4. Granularidad.

En la sección 2.4, cuando hablábamos de granularidad, decíamos que era importante para realizar un buen balanceado de la carga. Cuando entramos en sistemas cliente-servidor a través de Internet hay que tener en cuenta más factores que se pueden ver influidos si elegimos un tipo de granularidad u otra. El tamaño de la unidad de procesamiento marcará también el tamaño del mensaje que se enviará a través de Internet, que a su vez tomará un tiempo de envío a través del canal, también determinará cuanto tiempo tomará el procesado en el servidor (mensajes con más datos tardaran más tiempo en procesarse). En el caso de que haya un fallo habrá que reenviar el mensaje. Si el mensaje es muy pequeño puede que estemos malgastando mucho ancho de banda en cabeceras de HTTP con mensajes casi vacíos. Si el mensaje es muy grande el servidor tomará mucho (mientras está recibiendo el mensaje) antes de empezar a procesarlo.

Teniendo en cuenta todo lo anterior, determinamos que una granularidad en la que se toma como unidad de procesamiento un corte horizontal del órgano sigue siendo válida en un entorno cliente-servidor. El tamaño de mensaje oscila entre 1 y 4 megabytes y en un cliente con una conexión de 50Mb/s tarda menos de un segundo en realizar el envío, tiempo tras el cual el servidor puede empezar a procesar.

2.5.5. Tolerancia a fallos.

Como sabemos, Internet no es un canal perfecto y pueden ocurrir errores, por lo que hubimos de diseñar una política de tolerancia a fallos para protegernos de situaciones anómalas de la red o de nuestro propio software. Para simplificar el modelo determinamos que un servidor se ejecutaría

sobre un único *core* en un *host* remoto y podría procesar un único mensaje al mismo tiempo; en caso de que llegara un mensaje durante el procesamiento de uno previo se consideraría que habría una colisión y se daría al último en llegar como fallido devolviendo un código de error 500 al cliente. Cuando un cliente reciba un código de error 500 como respuesta a una petición, esperará un tiempo de gracia de cinco segundos antes de reenviar la petición para no sobrecargar el canal de comunicación.

2.5.6. Proveedores Cloud

Antes de que aparecieran las plataformas Cloud, el despliegue de los servicios WEB conllevaba una serie de retos como el desembolso inicial para la adquisición de equipos donde desplegarlos. Si preveías que tu aplicación iba a tener X usuarios simultáneos, adquirías una cierta cantidad de equipos para satisfacer esa demanda. Posteriormente puede que la mayor parte del tiempo tus usuarios reales sean X/5 por lo que tendrías el 80% de tu hardware haciéndose viejo sin ser utilizado. También podría pasar que tu aplicación tuviera más éxito del esperado y los usuarios fueran 3X, lo que inmediatamente colapsaría tus servidores y darías mal servicio, enfadando a muchos usuarios hasta que fueras capaz de adquirir y poner en marcha más hardware, momento en el cual posiblemente ya no te hiciera falta porque los usuarios se habrían ido a la competencia por las caídas del servicio.

Los servicios Cloud se postulan como una solución a este tipo de problema. Una empresa que quiera poner en marcha un servicio WEB puede contratar bajo demanda en tiempo real y a través de Internet recursos hardware, prácticamente ilimitados, que estarán inmediatamente disponibles para satisfacer la demanda del servicio permitiendo su **escalabilidad**. Además, los servicios Cloud nos ofrecen **elasticidad**, un usuario de estos servicios puede incrementar o decrementar la cantidad de recursos contratados de forma automatizada ajustándose en cada momento a la carga del sistema para pagar solo por el hardware necesario en cada momento. En nuestro caso particular QUIBIM S.L podría contratar servicios Cloud en función de su propia demanda interna y también podría hacerlo para ofrecer su algoritmo como SaaS a terceros.

Las plataformas Cloud son gigantes tecnológicos que ofrecen principalmente dos cosas: Infraestructura como servicio (IaaS) y Plataforma como servicio (PaaS). El IaaS consiste proporcionar acceso a recursos informáticos situados en un entorno remoto (Cloud), a través de una conexión pública, que suele ser internet. Los recursos informáticos ofrecidos consisten en hardware con unas características definidas como el tipo de procesador, la cantidad de disco duro, el tamaño de la memoria RAM, la velocidad de conexión, … En el caso de los PaaS, son servicios que ofrecen las plataformas Cloud para el soporte a las aplicaciones de sus clientes, por ejemplo, un PaaS puede ser un servicio de base de datos en Cloud con el cual el desarrollador puede montar

su aplicación sin preocuparse de configurar ni administrar la base de datos, simplemente usando dicho servicio.

Las tres principales plataformas Cloud son Amazon Web Services (AWS) [32], Microsoft Azure [33] y Google Cloud Platform (GCP) [34]. La pionera fue AWS y es la que va en cabeza tanto en usuarios como en la cantidad de servicios que ofrece. Microsoft está apostando fuerte por hacer que Azure esté al nivel de AWS aunque todavía le falta recorrido. Por su parte GCP fue la última en unirse a la carrera y aún no está al nivel de las otras dos. Cabe decir que han surgido muchas plataformas Cloud mucho más minoritarias pero que intentan hacerse un hueco en el mercado.

Para nuestro SaaS elegimos como plataforma Cloud a AWS porque su servicio de autoescalado nos es mucho más conveniente para nuestro tipo de aplicación que el del resto de plataformas. A continuación, hemos detallado que servicios Cloud hemos utilizado de AWS y como lo hemos hecho:

2.5.6.1. AMI.

Una AMI [35] no es más que una imagen de una máquina con un determinado software instalado que puede utilizarse para crear rápidamente nuevas instancias y ponerlas en ejecución con dicho software. Nosotros creamos una AMI con Ubuntu 14.04 instalado a la que le añadimos nuestras dependencias (Ceres Solver y libmicrohttpd) y nuestro algoritmo de manera que se iniciara automáticamente el servidor al iniciarse Ubuntu.

2.5.6.2. Load Balancer.

El servicio de Load Balancer (LB) [36][figura 7] nos ofrece un punto de entrada único a nuestro SaaS desde Internet al que se conectarán todos los clientes y será el encargado de repartir esas conexiones entre todas nuestras instancias de servidor activas. No está muy claro como el LB hace el reparto de las conexiones entre los servidores, pero por nuestra experiencia podemos decir que es algo parecido a una distribución *round-robin*.

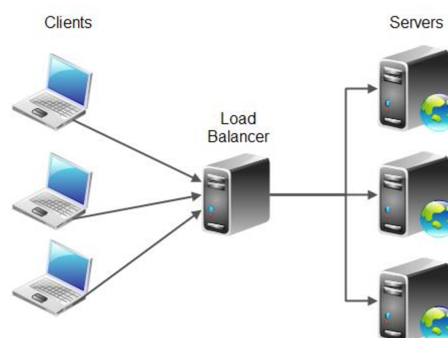

Figura 7: Esquema de un balanceador de carga.

2.5.6.3. Auto-Scaling Group

El grupo de autoescalado [37] es un servicio automatizado que, con ayuda de unas políticas de elasticidad que debemos diseñar, se encarga de controlar el estado del SaaS. Cuando crece la carga del sistema crea instancias de nuestro servidor a partir de la nuestra AMI añadiéndolas automáticamente al LB para su puesta en producción. Cuando la carga del sistema baja destruye instancias para ahorrar recursos [figura 8].

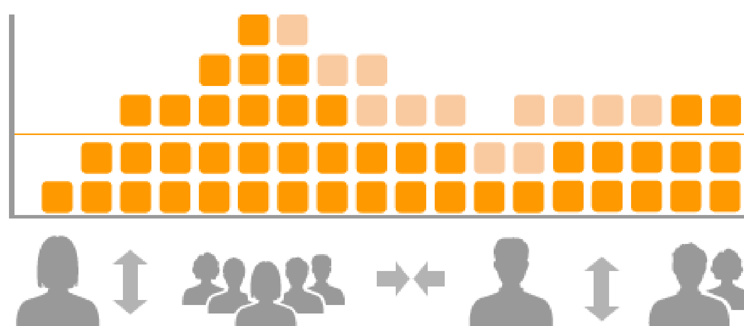

Figura 8: Representación del nivel de carga de un servicio Cloud en función del número de usuarios en cada instante.

2.6. Datasets e Infraestructura

Para la realización del análisis comparativo entre la solución propuesta en este TFM y la implementación original de QUIBIM S.L., se utilizaron un conjunto de diez volúmenes proporcionados por el HUPLF, que contenían información de casos reales sobre glándulas mamarias.

Para el estudio comparativo, los volúmenes fueron anonimizados para que no quedaran metadatos que pudieran relacionarlos con los pacientes a los que pertenecían, con el objetivo de preservar su derecho a la intimidad. La Tabla 1 muestra un resumen de las características principales del dataset:

Tabla 1. Esta tabla nos muestra las características de los volúmenes de prueba. La columna 2 indica el número total de voxeles en que se divide el volumen. Las columnas 3-5 indican el número de divisiones de los ejes X,Y y Z. La sexta columna indica el número de mediciones de la saturación de contraste que se han tomado para cada voxel.

Identificador	Nº voxeles	Tamaño eje X	Tamaño eje Y	Tamaño eje Z	Muestras por voxel
volume_00	6.291.456	256	256	96	7
volume_01	12.582.912	256	256	192	6
volume_02	12.582.912	256	256	192	5
volume_03	12.582.912	256	256	192	5
volume_04	12.582.912	256	256	192	5
volume_05	12.582.912	256	256	192	5

volume_06	12.582.912	256	256	192	6
volume_07	12.582.912	256	256	192	6
volume_08	12.582.912	256	256	192	5
volume_09	5.636.096	256	256	86	6

Para los trabajos en local hemos utilizado una máquina virtual de 6 cores modelo Intel I7 a 3400GHz, sistema operativo Ubuntu 14.04, 6 GB de memoria RAM, disco duro HDD y conexión a Internet de 300Mb/s de bajada y 30Mb/s de subida.

Para los trabajos en Cloud hemos utilizado instancias M1 medium de 1 core a 2000MHz desplegados en AWS, 3.75GB de RAM y disco duro HDD.

3. Resultados y Discusión.

El primer resultado son los algoritmos reimplementados del biomarcador que optimizan la versión original de Matlab. En segundo lugar, se presenta la arquitectura de la solución Cloud para la optimización e integración en el hospital del biomarcador que cumple los requisitos planteados en la sección 2.5. Finalmente se realiza un estudio comparativo entre las diferentes versiones de los algoritmos reimplementados y la solución Cloud frente a la solución Matlab inicial.

3.1. Algoritmos reimplementados

Pese a ser uno de los resultados más significativos de este trabajo, no se ha podido anexar o referenciar el código completo desarrollado dado que sería una vulneración de los derechos de autor de QUIBIM S.L.

3.2. Arquitectura de la solución Cloud

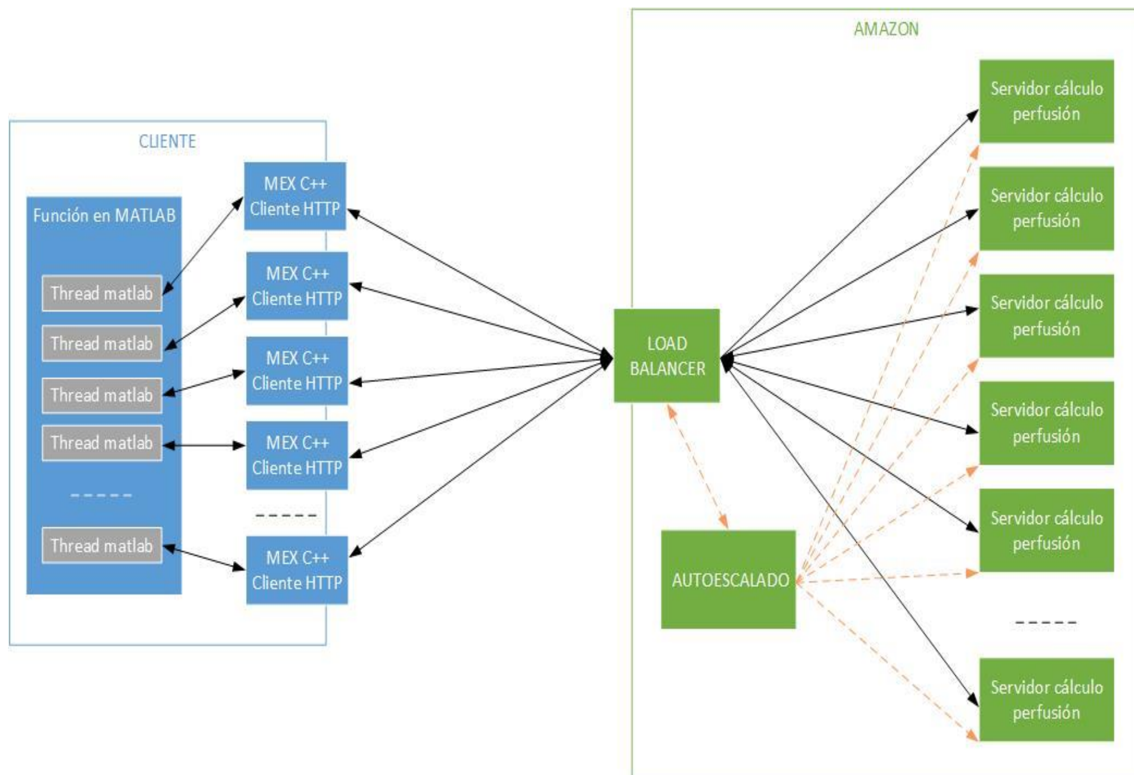

Figura 9: Esquema de la arquitectura Cloud del algoritmo.

Tal y como observamos en la figura 9, podemos ver los componentes que han conformado el cliente y el servidor. A continuación, se detalla cada uno de estos componentes. El cliente, en color azul, es una función en Matlab que recibe un volumen de datos para su procesamiento, crea un pool de threads, divide el volumen en cortes, con un bucle '*parfor*' va asignando los cortes a los threads disponibles, cada *thread* llama a una función MEX para enviar el corte que tiene asignado a través de Internet para su procesamiento en el Cloud y cuando acaban de procesarse

todos los cortes se devuelven al usuario los resultados. El Cloud, en color verde, consta de servidores que se encargan de realizar el cómputo, un balanceador de carga que reparte las peticiones entrantes entre los servidores disponibles y un grupo de autoescalado que mantiene activas las instancias de servidores necesarias para cubrir la demanda del servicio en cada instante.

3.3. Estudio y Análisis Comparativo

En esta sección se describe el estudio y análisis comparativo entre el algoritmo original de Matlab y las implementaciones optimizadas. En el caso de la solución Cloud, se ha realizado un estudio previo para medir los parámetros que permiten optimizar al máximo las ejecuciones en el Cloud.

La tabla 2 muestra los tiempos de ejecución del algoritmo original desarrollado en Matlab con los 10 volúmenes de prueba de que disponíamos:

Tabla 2. Tiempos de ejecución del algoritmo original. La columna dos indica la cantidad de voxeles que contiene el volumen, la tres el número de mediciones de la saturación de contraste que se han tomado para cada voxel, la cuarta el tiempo total de ejecución en segundos con el algoritmo original.

identificador	Número de voxeles	Muestras por voxel	Tiempo de ejecución en segundos
volume_00	6.291.456	7	91.815
volume_01	12.582.912	6	155.169
volume_02	12.582.912	5	149.873
volume_03	12.582.912	5	144.213
volume_04	12.582.912	5	135.465
volume_05	12.582.912	5	111.097
volume_06	12.582.912	6	162.884
volume_07	12.582.912	6	143.124
volume_08	12.582.912	5	183.034
volume_09	5.636.096	6	48.519

Como podemos observar, los tiempos de ejecución oscilan entre 13 y 50 horas de ejecución, en función del tamaño del volumen y de las características del mismo.

3.4. Reimplementación Secuencial Optimizada.

Después de implementar la primera modificación sobre el código, en la que en lugar de utilizar la función *lsqcurvefit* de MATLAB utilizábamos la librería Ceres Solver en C++, realizamos una prueba con el volumen_00 y obtuvimos un tiempo de ejecución de 768 segundos que corresponde a un *speedup* de 119,5. Solo utilizamos un volumen para esta prueba porque queríamos implementar más cambios antes de realizar test en profundidad. Este resultado es muy bueno y pone de manifiesto lo importante que es usar un lenguaje de programación adecuado para realizar cómputo de altas prestaciones.

3.5. Reimplementación paralela optimizada.

Tras la paralelización hicimos pruebas con el volumen_00 realizando el computo sobre cuatro cores y obtuvimos un tiempo de ejecución de 201 segundos lo que nos da un speedup de 3,82 con respecto al algoritmo secuencial reimplementado. Una vez más utilizamos solo un volumen para las pruebas porque solo queríamos una orientación del rendimiento de la mejora.

El speedup ideal sería 4, pero eso solo es en teoría, en la realidad todos los algoritmos paralelos tienen una sección de código que se ejecuta de forma secuencial y que no puede reducir su tiempo de ejecución mediante técnicas de paralelización. Esto lo explica muy bien la ley de Amdahl [38]. Aun así, 3,82 de 4 es un speedup muy bueno que indica que se pierde muy poco tiempo en secciones de código seriales.

3.6. Solución Cloud.

A continuación, se detallan un conjunto de pruebas realizadas sobre la implementación con la arquitectura Cloud con el objetivo de determinar los parámetros de configuración más adecuados para el propio Cloud.

3.6.1. Ratio clientes / servidores.

Una vez incorporada la arquitectura cliente servidor, quisimos determinar cuál era ratio clientes/servidores más adecuado para alcanzar un compromiso entre el tiempo total de ejecución y el nivel de ocupación de los servidores.

Para el siguiente test se ha determino que 8 fuera la cantidad fija de threads que el cliente podía tener en su *pool* durante todas las pruebas y hemos medido el tiempo de ejecución, para ejecuciones con distintas cantidades de servidores activos.

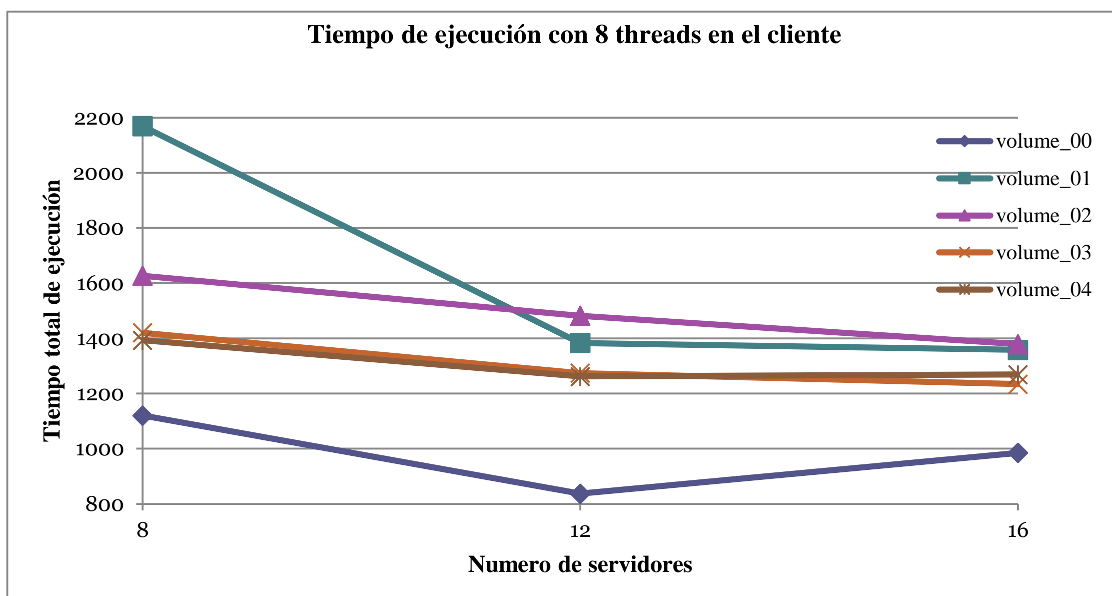

Figura 10: Pruebas con 8 threads en cliente.

En términos generales tener más servidores parece que mejora el tiempo de ejecución, aunque en la figura 10 podemos apreciar algunas discrepancias. Como vimos en el apartado de tolerancia a fallos se pueden producir colisiones cuando le llega una petición de trabajo a un servidor que está trabajando en otra previa. En nuestro despliegue el encargado de decidir cómo se reparten las peticiones entre los diferentes servidores disponibles es el LB. Como dijimos anteriormente, el LB tiene una política de reparto del trabajo que desconocemos pero que por nuestra experiencia parece ser *round-robin*.

En la figura 11 podemos apreciar la cantidad de colisiones que ocurrieron en cada una de las ejecuciones anteriores.

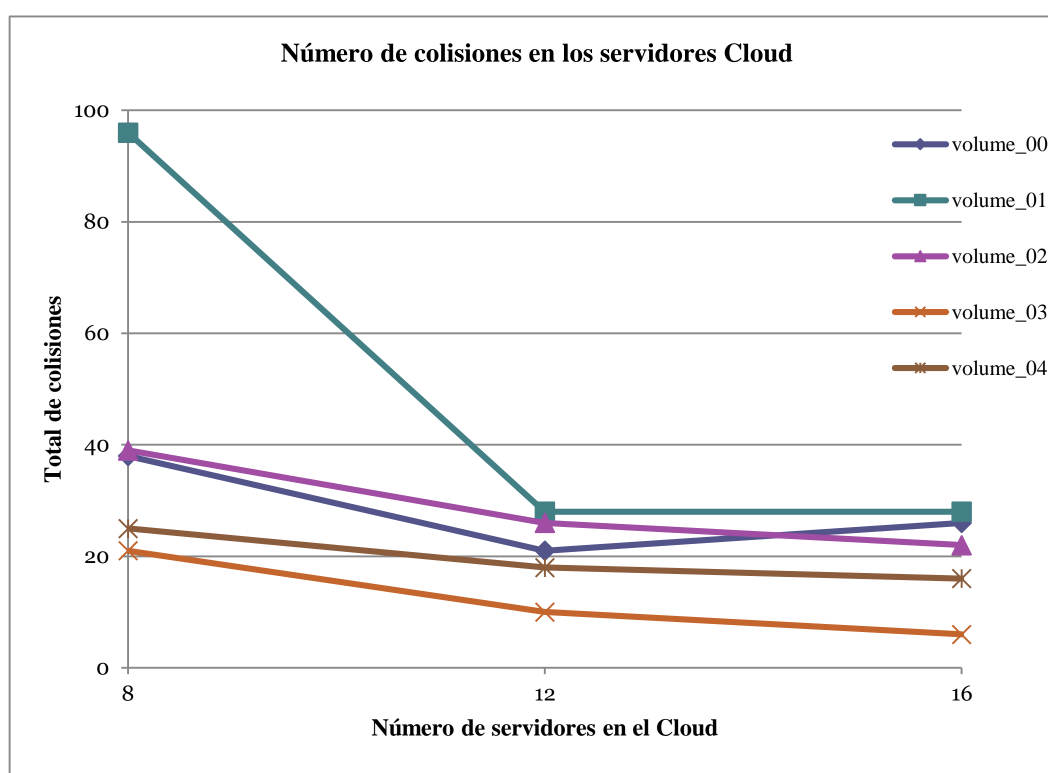

Figura 11: Evolución del número de colisiones con 8 threads en cliente.

Podemos comprobar que la cantidad de colisiones varia en las distintas configuraciones sin un patrón determinado, ello es debido a tres factores principalmente: el primero que el reparto que realiza el LB de las peticiones se ve afectado al variar el número de threads en el cliente y de instancias de servidores, también porque es indeterminado el orden en el que llegan las peticiones al LB y porque cada unidad de procesamiento enviada a los servidores tiene un tiempo de ejecución distinto. Todo lo anterior da lugar a repartos diferentes de la carga en cada ejecución.

Para acabar de tomar una decisión necesitábamos echar un vistazo a la figura 12 en la que pudimos comprobar la proporción de tiempo que estuvieron trabajando los servidores en cada ejecución.

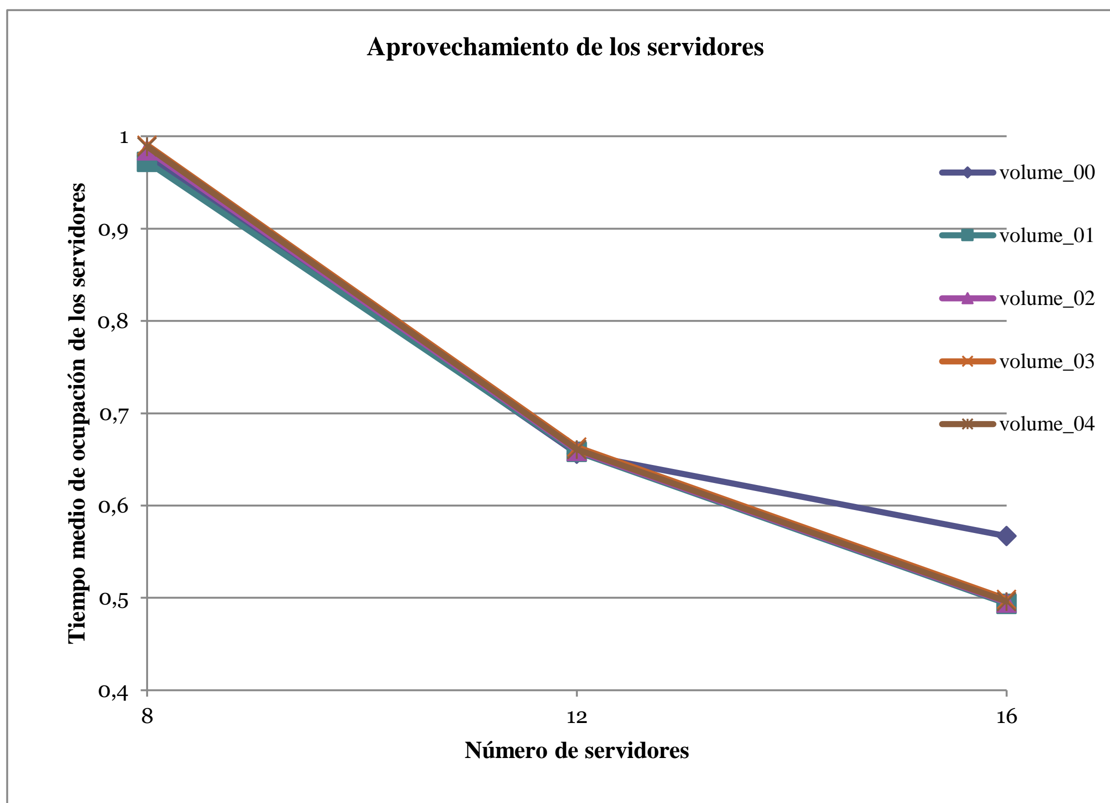

Figura 12: Nivel de ocupación de las instancias en el cloud.

Como podemos ver en la figura 12, conforme añadimos más instancias de servidores que threads tenemos en el cliente reducimos el nivel medio de ocupación de los servidores casi al 50%, esto se traduce en que estamos pagando casi el doble por recursos Cloud que no se están utilizando. Como la mejora en el tiempo de ejecución al añadir servidores extra no justificaba el hecho de pagar más para después desaprovechar la mitad de la potencia de cómputo, decidimos que nuestro caso óptimo sería tener la misma cantidad de threads en el cliente y de servidores en Cloud.

3.6.2. Test de escalabilidad

En este test, se comprueba cómo se comporta el cliente al escalar en el número threads. El objetivo del test es encontrar el cuello de botella y ver cómo mejorar los tiempos de ejecución. El host cliente es una máquina Linux con Ubuntu 14.04, 10 GB de RAM, 6 procesadores Intel I7 a 2.4 GHz y una conexión a Internet de 300Mbits/s. La figura 13 muestra los resultados obtenidos para los test realizados sobre 5 volúmenes con diferentes configuraciones.

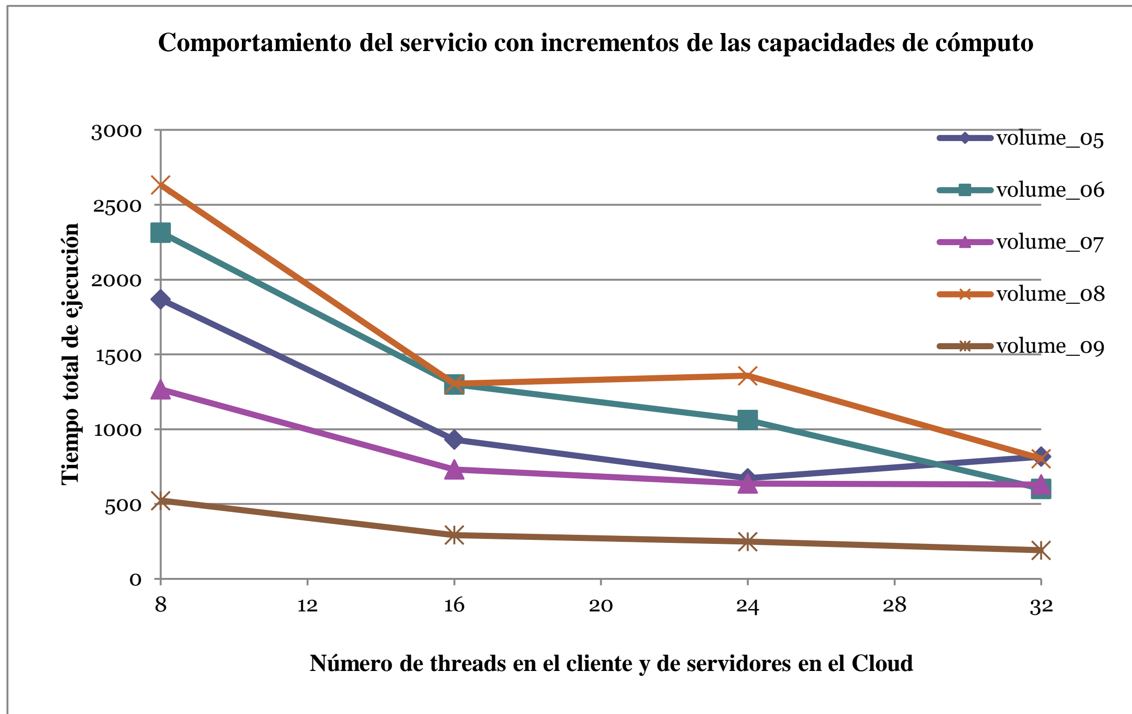

Figura 13: Test de escalabilidad del servicio.

Vemos como el algoritmo es escalable, conforme aumentamos el número de recursos que ponemos a trabajar en paralelo vamos reduciendo el tiempo de ejecución. El cuello de botella en nuestro cliente es la cantidad de memoria RAM; con más de 32 threads el sistema operativo comienza a hacer swaping con el disco duro y hace imposible el terminar la ejecución en un tiempo razonable.

En teoría, el grado máximo de paralelismo viene limitado por el número máximo de cortes horizontales del volumen a procesar. Para un volumen con 196 cortes se podrían lanzar 196 threads en el cliente y 196 servidores en el Cloud de manera que se pudieran calcular todos concurrentemente, en este supuesto obtendríamos el mejor tiempo de ejecución. Sin embargo, para poder obtener tanto paralelismo necesitaríamos un cliente mucho más potente para que pudiera satisfacer la demanda de recursos que se necesitan.

Queda por comprobar como evolucionaba el número de colisiones al incrementar el paralelismo. En la figura 14 podemos ver cuál fue el número de colisiones en las ejecuciones de este test.

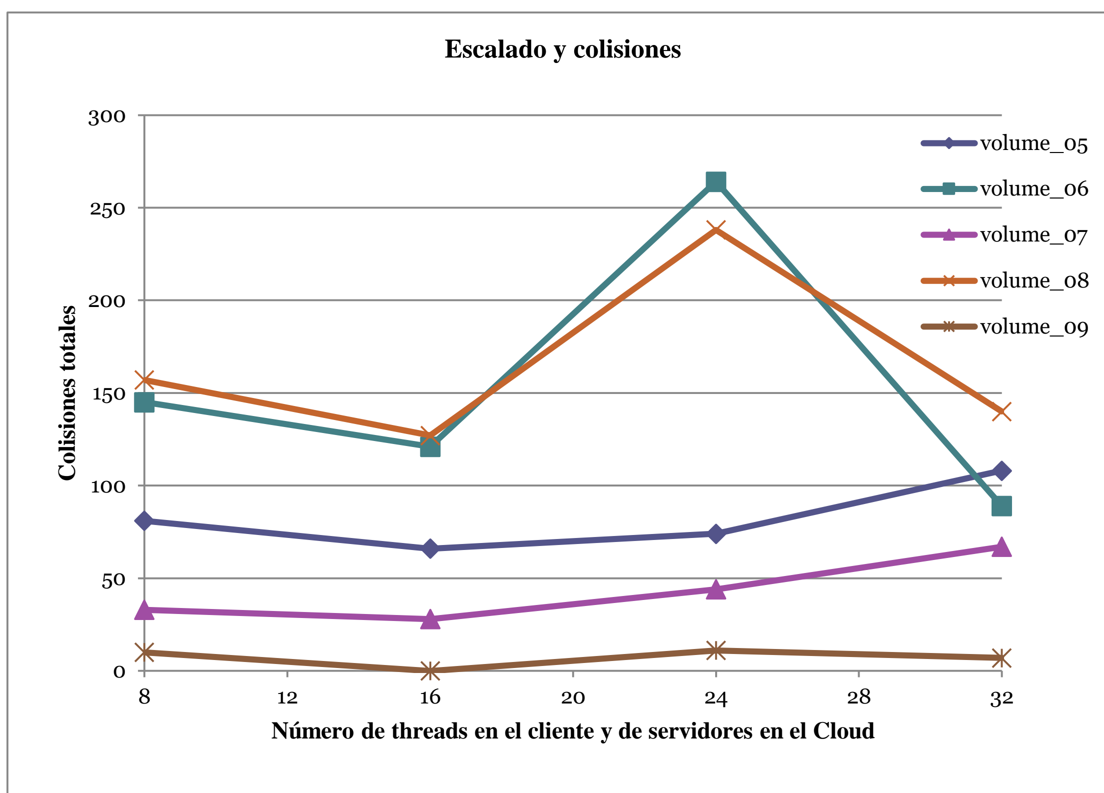

Figura 14: Evolución de las colisiones durante los test de escalado.

Como podemos apreciar, manteniendo igual el número de threads en el cliente y de instancias de servidores en el Cloud, no hay una correlación entre el número de colisiones y el aumento del nivel de paralelismo. Observamos fluctuaciones hacia arriba y hacia abajo sin un patrón definido, como comentamos en el capítulo anterior, esto puede ser debido al diferente reparto de la carga que ocurre en cada ejecución.

3.6.3. Algoritmo original vs solución Cloud optimizada

En la siguiente prueba queremos comparar el tiempo de ejecución original con el obtenido con nuestra optimización desplegada en Cloud. Para ello realizamos ejecuciones con los 10 volúmenes de que disponíamos, en cada ejecución el cliente generaba 32 *threads* y en el Cloud había 32 servidores ya activos y a la espera de peticiones.

Tabla 3. Esta tabla evalúa la mejora del tiempo de ejecución obtenido de procesar los volúmenes de test sobre el algoritmo original a procesarlos sobre la arquitectura Cloud. La columna 2 contiene los tiempos de ejecución en segundos del algoritmo original, la segunda los tiempos de ejecución en segundos en la ejecución en Cloud, la tercera el spedup obtenido y la cuarta el número de colisiones obtenidas en el Cloud.

id	Tiempo original	Tiempo Cloud	Speedup	Colisiones
volume_00	91.815	420	218,6	90
volume_01	155.169	632	245,5	130
volume_02	149.873	639	234,5	130

volume_03	144.213	443	325,5	50
volume_04	135.465	919	147,4	290
volume_05	111.097	817	135,9	108
volume_06	162.884	602	270,5	89
volume_07	143.124	631	226,8	67
volume_08	183.034	803	227,9	140
volume_09	48.519	191	254,0	7

Como vemos en la tabla 3, el tiempo medio de ejecución del algoritmo original era de 36,7 horas y el de nuestra optimización en Cloud de 10 minutos. El *speedup* medio fue de 228 lo que supone una mejora muy importante.

3.6.4. Elasticidad.

En nuestras pruebas anteriores partíamos de la base de que los servidores estaban activos y listos para trabajar, pero en un entorno de producción en Cloud esto no debe ser así. Solo debería haber activos los servidores necesarios para cubrir la demanda en cada instante. Para ello se habían de estipular algunas políticas de autoescalado que permitieran la gestión desatendida del conjunto de recursos Cloud utilizados.

Como mínimo necesitabamos especificar una política de aumento de recursos cuando el servicio recibiera más carga de trabajo y una de reducción de recursos conforme los servidores se quedasen sin trabajo que realizar. Los parámetros más significativos que había que configurar en el grupo de autoescalado son los siguientes:

- Mínimo número de instancias (Min).
- Máximo número de instancias (Max).
- *Health Check Grace Period* (HCGP), la cantidad de tiempo que el grupo de autoescalado espera tras crear un servidor antes de comenzar a hacer *health checks*.
- *Default Cooldown* (DC), número de segundos mínimo entre dos actuaciones del grupo de autoescalado.
- Politicas de decremento de servidores (PDS).
- Politicas de aumento de servidores (PAS).

3.6.4.1. Configuración 1

En la primera configuración introdujimos los siguientes parámetros:

- Min = 1
- Max = 32
- HCGP = 10 segundos

- DC = 60 segundos
- PDS: eliminar 4 instancias cuando la ocupación media de la CPU sea menor del 30% durante un periodo de 60 segundos.
- PAS: Añadir 8 instancias cuando la ocupación media de la CPU sea mayor del 80% durante un periodo de 60 segundos.

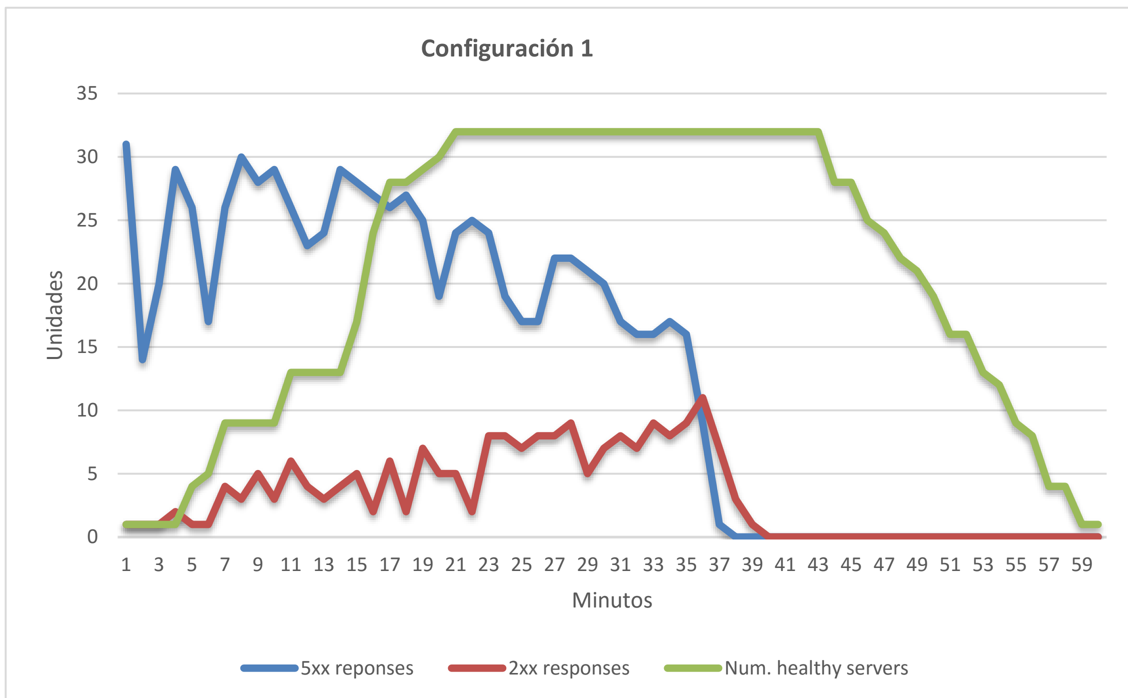

Figura 15: Estadísticas de la configuración 1.

En la figura 15, la línea azul indica el número de colisiones que sucedian, la línea roja el número de cortes computados y la línea verde el número de servidores activos en cada instante. Esta primera configuración tenia varias deficiencias como que el servicio tomaba demasiado tiempo en eliminar las instancias una vez acabado el trabajo y que el tiempo de ejecución fue de unos 40 minutos lo que nos parecía demasiado a la vista de las anteriores pruebas.

3.6.4.2. Configuración 2.

En la segunda configuración introdujimos los siguientes parámetros:

- Min = 1
- Max = 32
- HCGP = 10 segundos
- DC = 60 segundos
- PDS: eliminar 8 instancias cuando la ocupación media de la CPU sea menor del 30% durante un periodo de 60 segundos.

▪ PAS: Añadir 16 instancias cuando la ocupación media de la CPU sea mayor del 80% durante un periodo de 60 segundos.

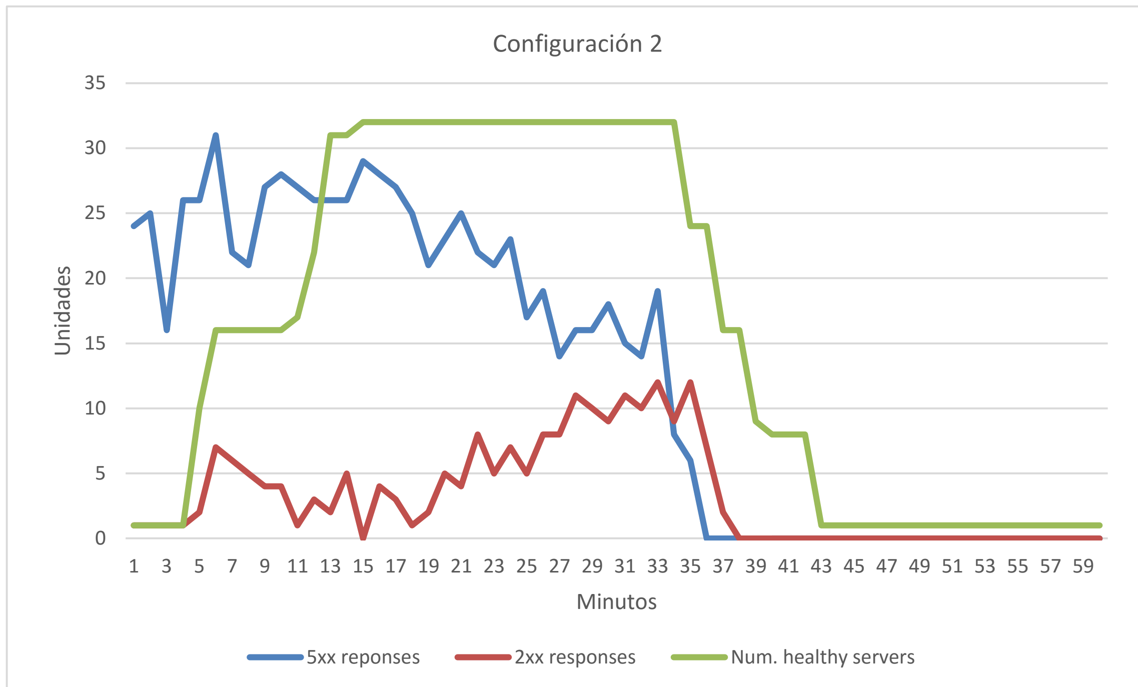

Figura 16: Estadísticas de la configuración 2.

En la figura 16 podemos ver que con esta configuración redujimos el tiempo que tardaban en liberarse recursos al terminar, pero todavía no teniamos una mejora importante en el tiempo de ejecución.

3.6.4.3. Configuración 3.

En la tercera configuración introdujimos los siguientes parámetros:

▪ Min = 1
▪ Max = 32
▪ HCGP = 10 segundos
▪ DC = 10 segundos
▪ PDS: eliminar 16 instancias cuando la ocupación media de la CPU sea menor del 30% durante un periodo de 120 segundos.
▪ PAS: Añadir 31 instancias cuando la ocupación media de la CPU sea mayor del 80% durante un periodo de 60 segundos.

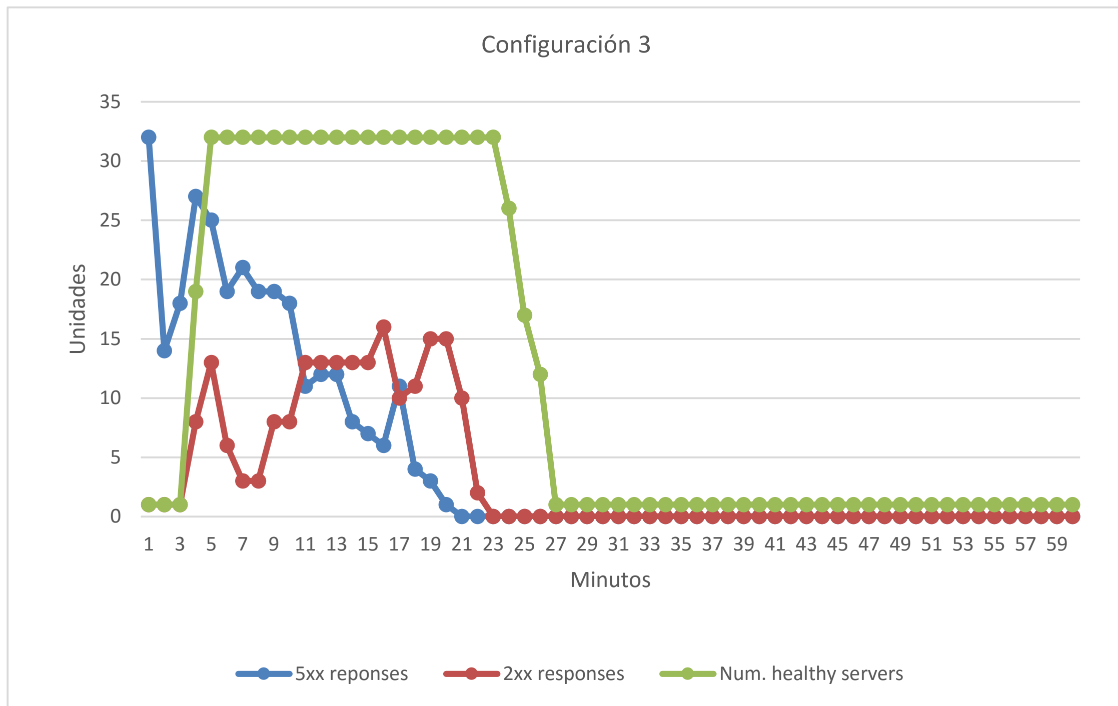

Figura 17: Estadísticas de la configuración 3.

En la figura 17 comprobamos que con esta última configuración conseguimos reducir también el tiempo de ejecución, que ahora es de 22 minutos. Este tiempo es mayor que el tiempo medio de 10 minutos que obteníamos en las pruebas en las que los servidores ya estaban preparados para trabajar, pero al utilizar el grupo de autoescalado, solo necesitaremos tener un único servidor activo cuando no haya trabajo que realizar.

4. Conclusiones

Matlab es solo para prototipar: En este trabajo hemos visto cuan interesante puede llegar a ser el uso de un lenguaje de programación adecuado en lugar de Matlab en programas de gran carga computacional. Matlab no es una herramienta recomendable para su uso en producción en la práctica clínica dada su ineficiencia.

Paralelizar más allá de los cores: El paralelismo local sobre los cores de las CPUs modernas está muy bien pero puede no ser suficiente para programas de cómputo masivo como el que hemos visto. Las plataformas Cloud nos ofrecen recursos 'ilimitados' para paralelizar nuestra carga de trabajo.

Granularidad y tolerancia a fallos: Para convertir un código monolítico en un servicio Cloud se debe dividir la carga de trabajo en bloques de un determinado tamaño, la elección de ese tamaño es muy importante ya que influirá en el reparto de la carga de trabajo, la duración, tamaño y cantidad de comunicaciones, la cantidad de fallos y la forma de recuperarse de ellos …

Escalabilidad y elasticidad: Estos son los pilares de todo servicio Cloud. Tu servicio debe satisfacer la demanda de tus clientes, un incremento de la demanda debe poder satisfacerse simplemente añadiendo más servidores al servicio. La cantidad de servidores debe adaptarse automáticamente a la demanda sin intervención del administrador.

Biomarcadores en Matlab: Este tipo de algoritmos son el ejemplo de como un programa puede pasar de utilizarse en un entorno meramente de investigación a poder utilizarse en producción en un Cloud, como un SaS, si se optimiza y se despliega sobre una arquitectura Cloud.

5. Referencias

[00]. Lindley Darden: **Recent Work in Computational Scientific Discovery**. In Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society 1997.

[01]. Roger D. Peng: **Reproducible Research in Computational Science**. Science 02 Dec 2011.

[02]. William L. Jorgensen: **The Many Roles of Computation in Drug Discovery**. Science 19 Mar 2004.

[03]. Imagen en oncología, Editorial Medica Panamericana. Yolanda Pallardó Calatayud, Antonio José Revert Ventura, José Cervera Deval y otros.

[04]. Picture Archiving and Communication System. http://searchhealthit.techtarget.com/definition/picture-archiving-and-communication-system-PACS.

[05]. Librería PETSc. https://www.mcs.anl.gov/petsc/.

[06]. MATLAB. http://es.mathworks.com/products/matlab/index.html.

[07]. M. Foracchia. **Detection of optic disc in retinal images by means of a geometrical model of vessel structure**. IEEE Oct 2004.

[08]. Rajesh C. Patil, Dr. A. S. Bhalchandra. **Brain Tumour Extraction from MRI Images using MATLAB**. International Journal of Electronics, Communication & Soft Computing Science and Engineering.

[09]. Jost, Gabriele, Jin, Hao-Qiang, anMey Dieter & Hatay, Ferhat F. **Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster**. NASA Technical Reports Server.

[10]. Yun He & H. Q. Ding. **MPI and OpenMP Paradigms on Cluster of SMP Architectures: The Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition**. Supercomputing, ACM/IEEE 2002 Conference.

[11]. E. Laure1 , S. M. Fisher2 , A. Frohner1 , C. Grandi3 , P. Kunszt4 , A. Krenek5 , O. Mulmo6 , F. Pacini7 , F. Prelz2 , J. White8 , M. Barroso1 , P. Buncic1 , F. Hemmer1 , A. Di Meglio1 , A. Edlund6 . **Programming the Grid with gLite**. COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY.

[12]. Michael Armbrust , Armando Fox, Rean Griffith , Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia. **A view of Cloud computing**. Communications of the ACM.

[13]. Quibim S.L. . http://quibim.com.

[14]. Roberto Sanz-Requena & Antonio Revert-Ventura & Luis Martí-Bonmatí & Ángel Alberich-Bayarri & Gracián García-Martí: **Quantitative MR perfusion parameters related to survival time in high-grade gliomas**. Eur Radiol.

[15]. Luis Martí-Bonmatí, Roberto Sanz-Requena, Angel Alberich-Bayarri. **Pharmacokinetic MR analysis of the cartilage is influenced by field strength**. European Journal of Radiology.

[16]. Formato NIFTI. http://nifti.nimh.nih.gov/nifti-1.

[17]. Función lsqcurvefit de Matlab. http://es.mathworks.com/help/optim/ug/lsqcurvefit.html.

[18]. Archive MEX. http://es.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html.

[19]. Librería DCMTK. **http://support.dcmtk.org/docs/.**

[20]. Librería VTK. http://www.vtk.org/.

[21]. Librería ITK. https://itk.org/.

[22]. Libreria MPFIT. https://www.physics.wisc.edu/~craigm/idl/cmpfit.html.

[23]. Librería GNU GSL. http://www.gnu.org/software/gsl/.

[24]. Librería Ceres Solver. http://ceres-solver.org/.

[25]. Librería OpenMP. http://openmp.org/wp/.

[26]. Librería Open-MPI, una implementación del estándar MPI. https://www.open-mpi.org/.

[27]. Wiki de la Librería Pthreads. https://en.wikipedia.org/wiki/POSIX_Threads.

[28]. Computación en la nube. https://es.wikipedia.org/wiki/Computaci%C3%B3n_en_la_nube.

[29]. Librería libcurl. https://curl.haxx.se/libcurl/.

[30]. Librería GNU libmicrohttpd. https://www.gnu.org/software/libmicrohttpd/.

[31]. Hypertext Transfer Protocol. https://www.w3.org/Protocols/.

[32]. Amazon Web Services. https://aws.amazon.com/es/

[33]. Microsoft Azure. https://azure.microsoft.com/es-es/.

[34]. Google Cloud Platform. https://Cloud.google.com/.

[35]. Amazon Machine Images.
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html.

[36]. Amazon Elastic Load Balancing. https://aws.amazon.com/es/elasticloadbalancing/.

[37]. Amazon Auto Scaling Group. https://aws.amazon.com/es/autoscaling/.

[38]. Ley de Amdahl. https://es.wikipedia.org/wiki/Ley_de_Amdahl.

ANEXO 1

PLANIFICACIÓN Y EJECUCIÓN DE TAREAS

	22/12/2015	23/12/2015 al 14/2/2016	15/01/2016	16/1/2016 al 7/2/2016	08/02/2016	9/2/2016 al 10/2/2016	11/02/2016	12/02/2016 al 16/5/2016	17/05/2016	18/5/2016 al 15/9/2016
1. Reunión presentación y objetivos										
2. Analisis y propuesta										
3. Presentación de la propuesta										
4. Primeras implkementaciones en C++										
5. Reunión primeros resultados										
6. Paralelización										
7. Reunión propuesta Cloud										
8. Implementación Cloud y test										
9 Presentación de resultados finales										
10. Redacción										

1. **22/12/2015**. Reunión en el HPULF con Quibim para el inicio del proyecto. En esta reunión Quibim expuso cual era el objetivo general, nos facilitó el algoritmo original, se establecieron los requisitos generales que aparecen en la sección 2.2 del TFM y dio al alumno un periodo de reflexión para desarrollar una propuesta de mejora.
2. **23/12/2015 al 14/1/2016**. Durante este tiempo el alumno examinó el código y diseño una propuesta de mejora basada en transformar a C++ el código principal.
3. **15/1/2016**. Reunión en el HPULF con Quibim. El alumno presenta la propuesta de transformar el código a C++ y Quibim la acepta.
4. **16/1/2016 al 7/2/2016**. Periodo en que el alumno busca librerías open–source para utilizar en el proyecto y realiza las primeras implementaciones en C++. Se realizan unos test para evaluar la viabilidad de la mejora.
5. **8/2/2016**. Reunión en el HPULF con Quibim. Se muestran los resultados del primer test de la reimplementación secuencial (sección 3.4) y se propone al alumno encontrar una solución adecuada para paralelizar el algoritmo y el estudiar la viabilidad de realizar una implementación Cloud.
6. **9/2/2016 al 10/2/2016**. Se sigue perfeccionando el código de cómputo, se evalúa el uso de threads para paralelizar el algoritmo en local y se diseña y testea una arquitectura cliente-servidor para validar la idea de desarrollar una arquitectura Cloud.
7. **11/2/2016**. Reunión en el HPULF con Quibim. El alumno expone los resultados de los test de paralelización (sección 3.5) y propone una arquitectura Cloud basada en los requisitos de la sección 2.5. Quibim está de acuerdo con la propuesta.
8. **12/2/2016 al 16/5/2016**. El alumno desarrolla la implementación en Cloud y realiza todas las pruebas necesarias para encontrar una configuración adecuada y realizar la evaluación final de la mejora terminada (sección 3.6).
9. **17/5/2016**. Reunión en el HPULF con Quibim. El alumno presenta los resultados y se propone la publicación de un artículo con el contenido de este proyecto.
10. **18/5/2016 al 15/9/2016**. El alumno redacta este trabajo e inicia la redacción del artículo. Además, Se envía una propuesta y es aceptada para exponer el contenido de este trabajo en el ESOI EuSoMII Annual Meeting 2016 que se realiza en octubre.