UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Paralelización Híbrida del Código AHF para el Post-Procesado de Simulaciones Cosmológicas

Trabajo Final de Máster

Máster Universitario en Computación Paralela y Distribuida

Universitat Politècnica de València

Autor
Fernando Campos del Pozo

Tutor
José E. Román

Co-Tutor
Alexander Knebe

Julio 2014

Agradecimientos

“El capital quiere hacernos creer que somos lo que vendemos. Pero somos lo que regalamos” (Jorge Riechmann)

Este trabajo está dedicado a mi padre, que tanto me enseñó. Gracias a su inmensa confianza en mí me enseñó a confiar en mí mismo. Gracias a tantas conversaciones y a su ejemplo, llevo una parte de él en mi forma de ser. Quizá sea la mejor herencia que pueda recibirse.

Dedicado también a mi madre, que con su ejemplo me ha dado grandes lecciones y con su apoyo cualquier reto parece abordable. Porque sin nuestras largas conversaciones después de la cena mi vida sería mucho más aburrida. Y porque sin su inmensa paciencia escuchando mis explicaciones sobre supercomputación y la materia oscura, no habría tenido a quien torturar.

A mi hermano, por no dejar de advertirme de lo dura y sacrificada que es la investigación, y aún así apoyarme siempre que lo he necesitado.

A Al, por todo el cariño y apoyo durante estos años, y por todos los momentos que hemos compartido juntos.

Quiero agradecer enormemente a José Román que aceptase la propuesta de dirigirme en este Trabajo Final de Máster, su inmensa ayuda durante la inspección y mejora de este software, y sus múltiples consejos y orientaciones. Mi agradecimiento también para Alexander Knebe, por resolverme tantas dudas, en persona y por e-mail, con infinita paciencia innumerables veces.

Un agradecimiento especial para Jose Luis Betí, técnico del Departamento de Sistemas Informáticos y Computación en la Universitat Politècnica de València, que me ha ayudado enormemente con la instalación de herramientas en el cluster kahan y con la utilización del mismo.

Gracias al grupo de investigación de Astrofísica y Cosmología del Departamento de Física Teórica de la Universidad Autónoma de Madrid por permitirme utilizar sus máquinas para desarrollar, probar y ejecutar el código desarrollado en este trabajo.

Gracias al equipo de herramientas de rendimiento del BSC, particularmente a Harald Servat que tantísimas preguntas me ha respondido, y al equipo de soporte del BSC.

Gracias a Troy D. Hanson por haber desarrollado su proyecto uthash y por haberlo compartido bajo una licencia de software libre. Y por mantenerlo vivo respondiendo a los usuarios de manera altruista.

Gracias a mis profesores del Máster Universitario en Computación Paralela y Distribuida, incluidos los profesores de las asignaturas del máster del DISCA, por su calidad docente y humana y por haberme enseñado tanto.

Gracias a José Flich por ayudarme a pensar en cómo reconvertir un recorrido recursivo en uno en amplitud y su relación con los algoritmos de broadcast.

Gracias a José del Peso y a Gustavo Yepes por darme la oportunidad de trabajar en el Departamento de Física Teórica de la Universidad Autónoma de Madrid, y haberme permitido descubrir el apasionante mundo de la física.

Un enorme agradecimiento por orientarme con los conceptos de la física, y por su amistad, a Sergio, a mis compañeros del despacho 512 (Germán, Pablo, Clara, Antonio y un largo etcétera), a mis compañeros del 313 (Jesús, Arianna, Federico y Edoardo) y, en general, a todos los compañeros del DFT y del IFT.

Resumen

En esta memoria expondremos el trabajo realizado para paralelizar y optimizar el rendimiento del código AHF, dedicado a identificar estructuras cosmológicamente relevantes en los resultados de simulaciones cosmológicas.

Durante los últimos años, el desarrollo y aumento de la capacidad de cómputo de los recursos computacionales, en concreto de los supercomputadores, han ampliado los horizontes de la investigación científica, de forma muy significativa en lo que a simulaciones numéricas se refiere. Estas nuevas herramientas son enormemente costosas económicamente, tanto su adquisición como su mantenimiento, pero también es fundamental tener en cuenta sus inmensos requerimientos de energía en un momento histórico de escasez de recursos energéticos.

La optimización de los códigos que se ejecutan en estos supercomputadores son por tanto doblemente importantes: pueden permitirnos hacer más y mejor ciencia, pero además podemos lograr reducir la cantidad de recursos computacionales necesarios reduciendo el tiempo de ejecución y aumentando a la vez la productividad científica. Esta optimización es cada vez más compleja debido, por una parte, a la gran complejidad de los códigos científicos, y por otra, a la creciente dificultad de desarrollar software que aproveche el gran número de nodos disponibles, la gran cantidad de *cores* presentes en cada nodo y la frecuente disponibilidad de aceleradores hardware.

Nuestro objetivo será, partiendo de este código ya paralelizado con MPI, analizar su diseño algorítmico y su rendimiento y, haciendo las modificaciones necesarias, aprovechar, mediante la utilización de OpenMP, la presencia de múltiples *cores* por nodo de los supercomputadores donde suele ejecutarse para reducir su tiempo de ejecución.

Índice

1. Introducción

La astronomía, la astrofísica, la cosmología y el uso de la simulación computacional como herramienta clave

A lo largo de la Historia, el ser humano se ha interesado por las leyes que rigen el movimiento de los objetos celestes que se observan desde la Tierra. Restos prehistóricos, como Stonehenge (1), datado en el siglo XX a.C., muestran conocimientos astronómicos por la exactitud de la proyección del Sol sobre esta construcción en el solsticio de verano. El conocimiento astronómico ha sido fundamental para avances de la Humanidad tan básicos como la capacidad de medir el tiempo, el establecimiento de un calendario anual y todas las implicaciones que estos avances suponen, por ejemplo, en la agricultura.

Tenemos constancia de que ya en la civilización griega se realizaron los primeros cálculos físicos relacionados con las estrellas y los planetas, mediciones de la distancia de la Tierra a la Luna y al Sol, y se propusieron ya modelos heliocéntricos.

El modelo geocéntrico fue propuesto por Aristóteles y ampliado por Ptolomeo (2), asumido e impuesto por creencias e imposiciones religiosas. Durante el Renacimiento fue cuando aparecieron los trabajos heliocéntricos de Copérnico, Galileo y Kepler (3), fuertemente perseguidos y/o intimidados por los tribunales de la Inquisición, y cuando sus teorías obtuvieron una notable aceptación. Finalmente Newton, basado en el trabajo de los científicos anteriores, descubrió la ley de la gravitación universal, convirtiéndose en el nuevo modelo aceptado (4).

A finales del siglo XIX, gracias a los descubrimientos de la descomposición en frecuencias de la luz del Sol y su correspondencia con las frecuencias de emisión de los elementos, se pudo comenzar a estudiar la composición de los cuerpos celestes (5). Avances espectaculares en el campo de la física permitieron comprender fenómenos fundamentales para los modelos cosmológicos actuales, que han incrementado el conocimiento actual sobre el origen y la evolución del universo. La medición del *redshift* (o corrimiento al rojo) (6) de la longitud de onda de las observaciones de luz y de otras radiaciones electromagnéticas, se corresponde con el efecto Doppler (velocidades relativas entre el emisor y el receptor) y permiten confirmar la expansión del universo, relacionada con la energía oscura.

Aun quedando fuera del alcance de este trabajo -y de mi conocimiento-, es necesario citar la teoría inflacionaria o de inflación cósmica, la cual postula que el universo, en sus momentos iniciales, sufrió una rápida expansión. En el periodo anterior solamente se producían fluctuaciones cuánticas y posteriormente comenzaron a formarse las partículas subatómicas, se produjeron fusiones nucleares y finalmente comenzaron a emitirse fotones capaces de escapar del alto potencial gravitatorio. Este postulado, planteado en la década de los 80 del pasado siglo, resuelve ciertos problemas de la teoría del BigBang. Experimentos actuales, que miden las ondas gravitacionales primordiales parecen confirmar la teoría de inflación cósmica (ver Figura 1 y 2) (7) (8) (9).

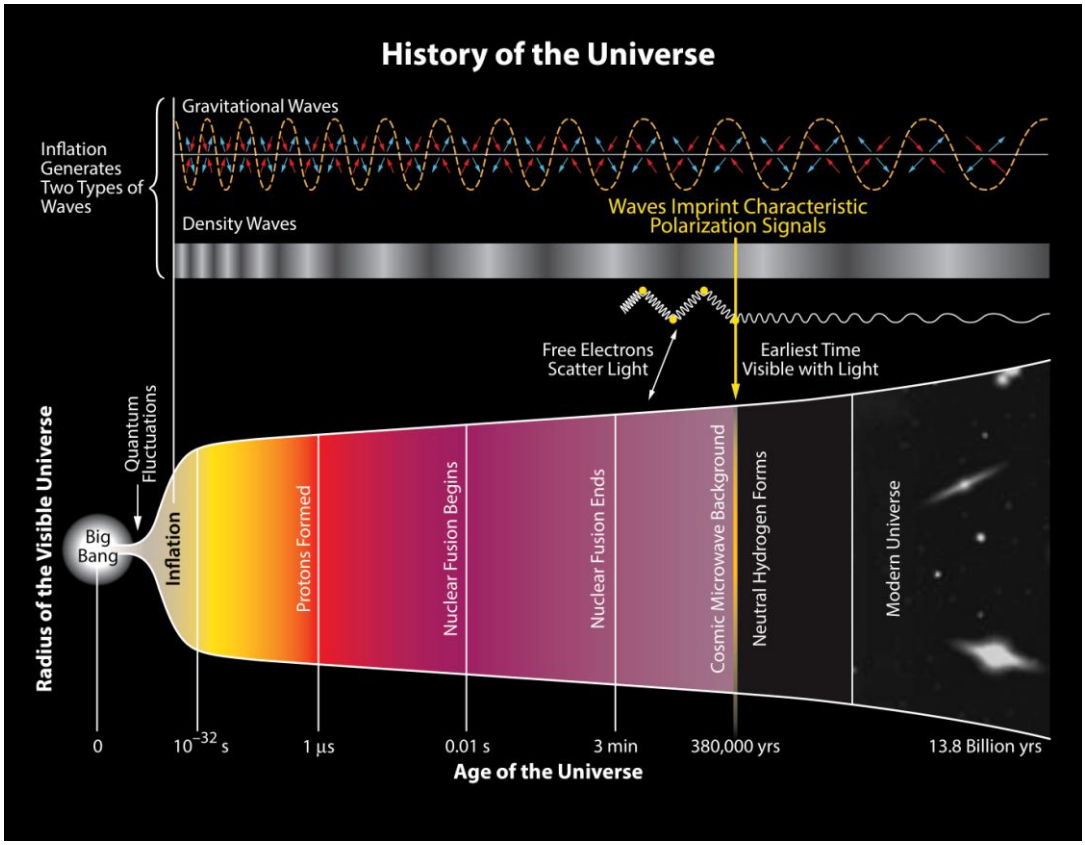

Figura 1. History of the Universe. Credit: Proyecto BICEP2, financiado por la NSF

Por otra parte, merece una mención especial el descubrimiento de la radiación cósmica de fondo (*CMB*, C*osmic Microwave Background* en inglés) a mediados de la década de 1960, pues supuso una de las principales pruebas de la teoría del Big Bang (ver Figura 5). Esta radiación, presente en todo el universo, sería el resto electromagnético posterior a la inflación cosmológica.

En la cosmología, dedicada al estudio del universo, su origen, su evolución y su estructura, existen como veremos varias dificultades añadidas.

Para el avance de las distintas ramas de la ciencia siempre se han diseñado experimentos que pudiesen probar o refutar una teoría. El avance de la ciencia sin la experimentación sería imposible. Ciertos objetos de investigación son más complejos que otros: el experimento que demuestra el principio de Arquímedes parece estar al alcance de cualquiera que disponga de un recipiente con un líquido y las herramientas para medir el volumen y el peso del elemento a sumergir y del fluido desplazado. En cambio otras áreas, como la física de partículas, han requerido la construcción de instrumentos de medición mucho más complejos como los aceleradores de partículas (quizá el más conocido sea el LHC del CERN (10)) o los detectores de neutrinos (probablemente menos conocidos, como el IceCube, situado en el Polo Sur (11)).

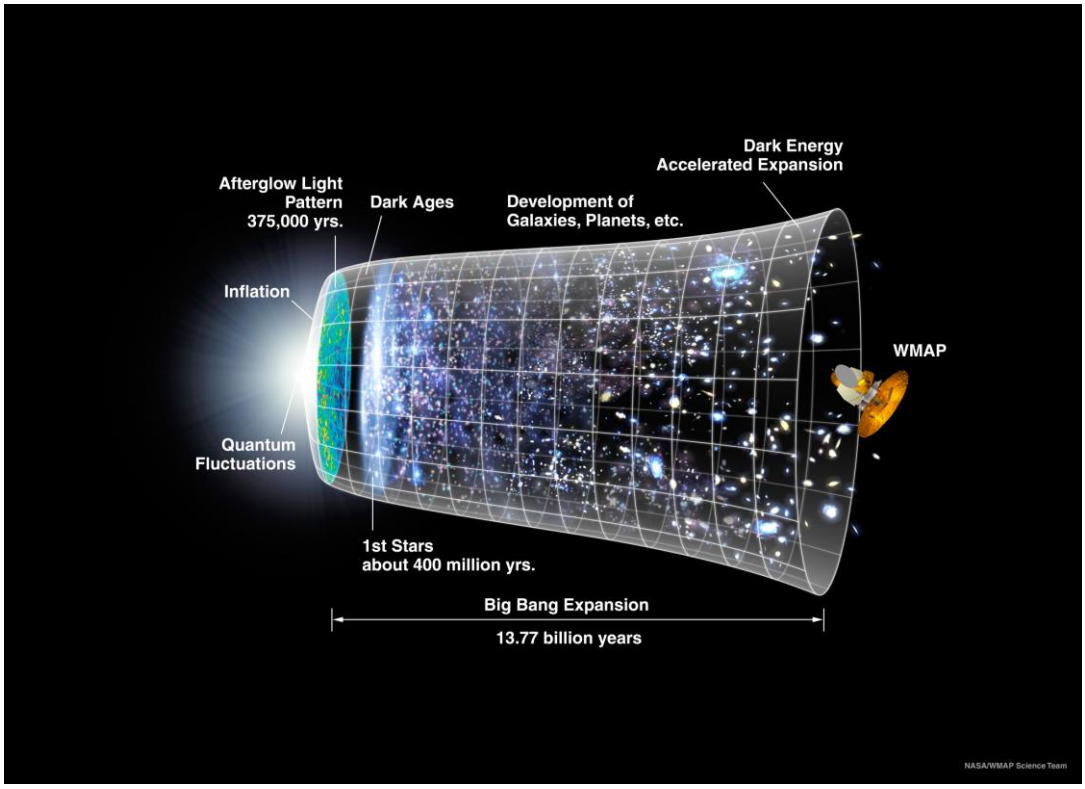

Figura 2. Timeline of the Universe. Credit: NASA / WMAP Science Team

En cambio, en cosmología, parece evidente la imposibilidad de realizar experimentos que repliquen un universo para distintos posibles escenarios (teorías alternativas a la relatividad general de Einstein, por ejemplo). Además, la cosmología (y también la astrofísica) se mueve en unas escalas de tiempo y espacio que dificultan enormemente la realización de medidas contrastables. Al observar con sofisticados telescopios objetos a distancias en el rango de los años luz, solamente seremos capaces de observar lo que ocurrió en el pasado. Sin ir tan lejos, la luz que recibimos del Sol tarda 8 minutos en alcanzar la Tierra: todos los fenómenos que observamos en el cielo corresponden al pasado. Si, por ejemplo, pretendiésemos observar nuestra galaxia, la Vía Láctea, que cuenta con un diámetro medio aproximado de 100.000 años luz, y fuésemos capaces de situarnos a observar o enviar un telescopio a uno de sus extremos, solamente seríamos capaces de ver el estado en el que se encontraba el otro extremo 100.000 años atrás.

Siguiendo el razonamiento anterior, cuanto más lejos miremos con un telescopio estaremos observando objetos de épocas anteriores del universo. Y lo más lejos y a la vez más atrás en el tiempo que podremos observar, siguiendo la teoría del Big Bang, sería la etapa posterior a la explosión inicial, una vez que los átomos comenzaron a formarse y los fotones pudieron "escapar". Eso es precisamente lo que se observa con la radiación cósmica de fondo: el rastro más antiguo que se puede detectar. Esta radiación, detectable en cualquier dirección de observación, contiene diferencias en los valores de temperatura y densidad. Estas *anisotropías* o irregularidades son la base para considerar la distribución inicial del universo que daría origen a la formación de estructuras de gran escala como la distribución de galaxias.

La astrofísica observacional, por tanto, permite descubrir las estructuras que existían en épocas anteriores del universo, buscando objetos más lejanos, y probar o refutar así las teorías. Pero sin la posibilidad de realizar experimentos, estamos limitados a una única realización: nuestro universo.

Otra dificultad añadida, por si lo anterior no era suficiente, es el hecho de que con los telescopios podemos observar las ondas electromagnéticas emitidas por la materia ordinaria que conocemos (bariónica). Las teorías aceptadas en la actualidad indican que este tipo de materia, que interacciona electromagnéticamente, compone aproximadamente el 5% de la materia-energía del universo. El otro 25% estaría formado por materia oscura, y el 70% restante por energía oscura (ver Figura 3).

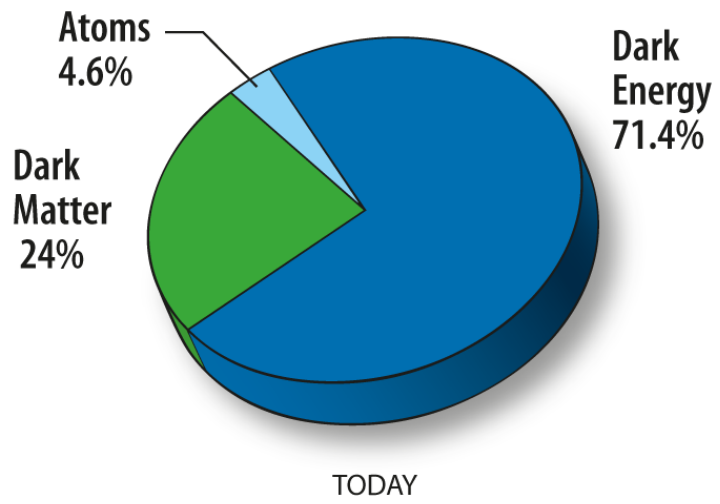

Figura 3. Universe Content - WMAP 9yr Pie Chart. Credit: NASA / WMAP Science Team

El hecho de que la materia que conocemos represente una fracción tan pequeña del total, y la necesidad de la existencia de la materia oscura, se deducen de la observación de las irregularidades en la radiación cósmica de fondo, así como en el estudio de las velocidades de rotación de galaxias. En este último caso, la energía cinética de ciertas estructuras, debida a la velocidad de rotación, son muy superiores al potencial gravitatorio generado por los objetos que somos capaces de observar, por lo que una de las posibles justificaciones para que se produzca esa ligadura gravitacional -hay otras teorías alternativas- sería la existencia de un tipo de materia que interactúe gravitatoriamente con una cantidad de masa significativa, pero no de manera electromagnética, ya que no podemos observarla. Al no emitir radiación electromagnética no se puede observar con telescopios, pero debe existir para justificar fenómenos conocidos y observados como las lentes gravitacionales (12) o el Bullet Cluster (13) (14). Otra característica particular de la materia oscura es que al no interactuar electromagnéticamente se pueden ignorar sus colisiones, a diferencia de lo que ocurre con la materia bariónica.

Tras revisar las inmensas dificultades que la astrofísica afronta para avanzar en la comprensión del origen, evolución y estructura del universo, la inmensa escala en la que se mueven sus

cálculos y la complejidad de la resolución de los mismos, debería parecer evidente, para cualquier científico computacional con conocimientos de supercomputación, la pertinencia del uso de las simulaciones cosmológicas con el soporte de la supercomputación. Esta herramienta permite afrontar el desafío de manera casi experimental, permitiendo comparar las teorías cosmológicas implementadas mediante simulaciones con las observaciones realizadas con telescopios.

Algunas de las mayores y más recientes simulaciones cosmológicas son Millenium Run (15), Bolshoi (ver Figura 4) (16) y Multidark (17) (18), las cuales han ido aplicando los descubrimientos realizados por las misiones espaciales WMAP5 y Planck en los parámetros de la física simulada. Los resultados de estas simulaciones están disponibles para la comunidad científica a través de internet. Las dimensiones de las simulaciones oscilan entre un tamaño de la caja de entre 250 y 2500 Megaparsecs y el número de partículas simuladas es de 2048^3 (8.5 miles de millones) y 3840^3 (56 miles de millones). Estas simulaciones y su post-procesado se ha realizado en supercomputadores de todo el mundo como Pleiades (NASA, EEUU), Supermuc (LRZ, Alemania), Marenostrum (BSC, España) y Juqueen (JSC, Alemania) (19).

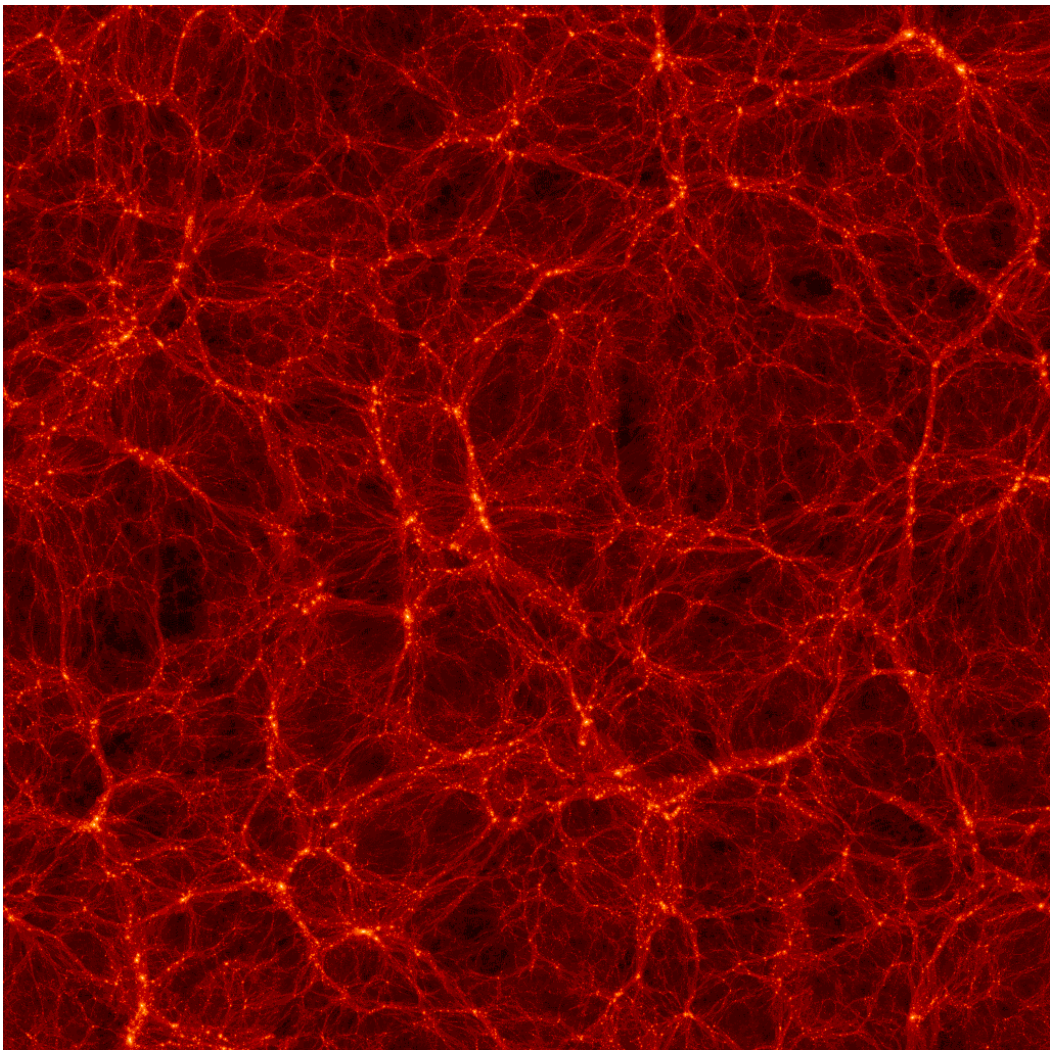

Figura 4. A slice of the whole 250Mpc box of Bolshoi. Credit: Made by Stefan Gottlober (AIP) with IDL

El software que permite realizar simulaciones cosmológicas está basado en la resolución de un problema de N-cuerpos (20) (21), en el que cada una de las partículas participa en la fuerza gravitacional sufrida por todas las demás. En la actualidad la escala en la que se ejecutan estas simulaciones es de miles de millones de partículas de materia oscura, cuya masa se mide en miles de masas solares, y las distancias se miden en miles y millones de parsecs. El parsec, unidad de longitud utilizada en astronomía, equivale aproximadamente a 3 años luz (22). Algunas simulaciones cosmológicas pueden incluir también partículas que representan gas y estrellas, debiendo introducir dinámica de fluidos en los cálculos, lo que aumenta aún más la complejidad.

Tratando de simplificar por brevedad, y porque mis conocimientos en la materia son muy limitados, podríamos asumir que una simulación cosmológica consiste en una caja que contiene un conjunto de partículas distribuidas según unas condiciones iniciales. Estas condiciones iniciales están basadas en las observaciones y análisis de la radiación cósmica de fondo (ver Figura 5) que se han llevado a cabo en misiones espaciales como WMAP (NASA, 2001) (23) y Planck (NASA y ESA, 2009) (24).

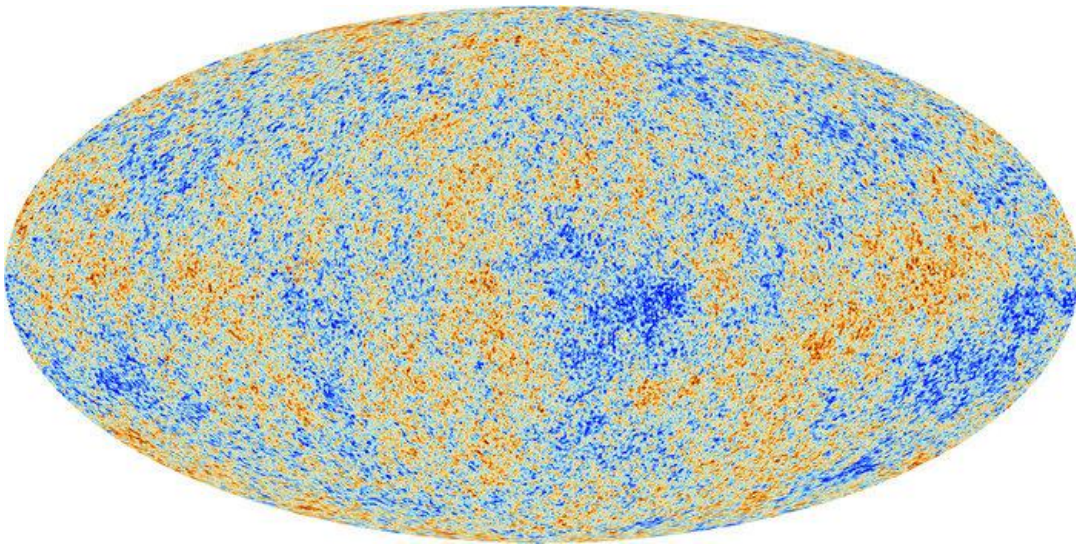

Figura 5. The anisotropies of the Cosmic microwave background (CMB) as observed by Planck. Credit: ESA and the Planck Collaboration, 2013

Deben cumplirse también condiciones de periodicidad en sus extremos, de manera que dos partículas próximas a dos caras opuestas de la caja, estarán próximas y se atraerán hacia el exterior de la caja. De no imponerse esta condición, la gravedad concentraría todas las partículas hacia un punto central, pero al simular una porción del universo no podemos asumir que no hay partículas fuera de la caja. Imponiendo condiciones de periodicidad se resuelve este problema.

Otro problema que resuelven estas simulaciones es el caso en el que dos partículas se encuentren muy próximas. Estos casos provocarían una fuerza de atracción extremadamente alta, ya que la fuerza gravitatoria aumenta con la inversa del cuadrado de la distancia. Si la distancia tiende a cero, la fuerza tenderá a infinito. Este problema suele resolverse ignorando la

fuerza gravitatoria entre partículas cuya distancia sea menor a un cierto umbral (25). Como puede intuirse, las complicaciones de implementación no son triviales.

Para hacer computacionalmente viable la simulación de enormes cantidades de partículas existen métodos más eficientes de aproximación de los cálculos. Por ejemplo, el algoritmo Barnes-Hut (26) (27), consistente en el refinamiento adaptativo para la subdivisión de la caja mediante la generación de un octree, esto es, un árbol en el cual cada nodo tiene 8 nodos hijos, correspondientes en este caso a la subdivisión de un cubo por la mitad en las tres dimensiones del espacio. Al contar con esta estructura, podremos evitar calcular las fuerzas de las N partículas con las otras N-1, pudiendo asumir que la fuerza ejercida por todas las partículas que pertenecen a un nodo del octree es equivalente a la que generaría una hipotética partícula cuya posición fuese el centro de gravedad del nodo y su masa la suma de las de las partículas que contiene. La reducción del coste computacional pasa a $O(n \log(n))$ desde el coste cuadrático ($O(n^2)$) que supone el cálculo directo entre cada par de partículas.

Otros algoritmos que permiten acelerar este tipo de simulaciones se basan en la generación de una malla sobre la distribución de partículas (*particle mesh*) (28) para cuyos vértices se calcularán los potenciales gravitacionales que afectarán a cada partícula, reduciendo así el coste computacional. Variaciones de este método permiten hacer cálculos entre las partículas cuando están próximas y utilizar la malla cuando las distancias son suficientemente grandes como para asumir los errores de cálculo (29).

Quizá el software más conocido y ampliamente utilizado para realizar este tipo de simulaciones es GADGET (30). Desarrollado inicialmente por el profesor Volker Springel como parte de su proyecto de doctorado bajo la supervisión de Simon White, la primera versión fue publicada en Marzo de 2000. La versión actual, GADGET-2, fue publicada en 2005 e incluye un método Tree-PM que combina los dos métodos expuestos anteriormente. Soporta la simulación de partículas de gas mediante el método SPH (31). Hace uso de las librerías para cálculo científico GSL (32) y FFTW (33), empleada esta última para el cálculo de las trasformadas de Fourier necesarias al trabajar con la malla de partículas. También incluye la posibilidad de utilizar HDF5 (34) para la generación de los ficheros con los datos de salida.

Independientemente del método y el software empleado para realizar la simulación cosmológica, el resultado será la posición y velocidad (junto con otros parámetros físicos fuera del alcance de este trabajo) de cada una de las partículas de la simulación en cada uno de los instantes en los que se genere un *snapshot*. Estos resultados nos permitirán obtener información sobre las regiones en las que se han aglutinado las partículas y sobre las estructuras que han formado. Pero la obtención de esa información no será producto de la simulación, sino del post-procesado que se realice sobre los resultados de la misma.

El último concepto astrofísico que debemos abordar para entender este trabajo es el de halo y en concreto el de halo de materia oscura. Un halo podría definirse como la región que engloba una estructura en el espacio. En esta región se cumplen ciertas propiedades comunes a todos los objetos contenidos en ella, quizá la más evidente sea estar ligados gravitatoriamente. En el caso de los halos de materia oscura, se trataría del hipotético conjunto de materia oscura que engloba el disco galáctico y se extiende más allá de los objetos visibles. No puede observarse por la naturaleza de la materia oscura, pero su presencia puede deducirse por el efecto que

provoca en el movimiento de las estrellas y los gases. La dificultad para analizar estas estructuras aumenta cuando consideramos que un halo de materia oscura puede contener a su vez halos internos (sub-halos) en los que, aumentando la resolución podremos observar subestructuras de menor tamaño. Identificar cuál o cuáles de estas subestructuras dominan gravitatoriamente (host-halos) es fundamental para estudiar su evolución.

Existe multitud de software que implementa este post-procesado y que trata de detectar los halos de materia oscura que se generan durante la evolución de la simulación. Cada uno se basa en distintos parámetros, y más frecuentemente en combinaciones de estos. Algunos de los más ampliamente utilizados son los siguientes (35):

- FOF (Friends-Of-Friends): Basándose en una distancia crítica (*linking length*) entre partículas, considera que todas las partículas separadas por esta distancia entre ellas pertenecen a un mismo grupo con características comunes. Estos grupos serían candidatos a formar un halo de materia oscura. Suelen realizarse análisis con distintos valores de distancia crítica para detectar estructuras de distintos tamaños.

- BDM (Bound Density Maximum): El principio de este método consiste en calcular la densidad de una cierta región esférica en la que existe un cúmulo de partículas y progresivamente aumentar el radio de la región disminuyendo la densidad hasta un valor mínimo umbral. El conjunto de partículas que se encuentran dentro de la región será candidato para formar un halo. Se suelen descartar aquellas partículas cuya velocidad es superior a la "velocidad de escape", es decir, aquellas cuya energía cinética es superior al potencial gravitatorio. Se asume que, en los siguientes *snapshots*, estas partículas se habrán alejado de la región. Aún así, su masa se debe considerar para el cálculo del potencial gravitatorio.

- AHF (Amiga Halo Finder): Este código de post-procesado de simulaciones cosmológicas, en el que se centra este trabajo, realiza un refinamiento adaptativo de la caja de la simulación, buscando cúmulos de densidad, es decir, regiones en las que el número de partículas sea mayor. La base para realizar la búsqueda de manera adaptativa es un número de partículas umbral (*refinement criterion*) que determinará si se debe seguir refinando cada una de las regiones con cúmulos de partículas. Este método completamente adaptativo permite detectar halos, sub-halos y subestructuras hasta niveles de refinamiento limitados únicamente por el criterio de refinamiento (36).

Como complemento necesario a estos métodos existe otro tipo de software, comúnmente llamado *MergerTree*, que, comparando las estructuras encontradas en los diferentes *snapshots*, trata de trazar la evolución de las estructuras a lo largo de la evolución de la simulación.

1.1. Objetivos

El objetivo principal de este trabajo es reducir el tiempo de ejecución del código AHF para el post-procesado de simulaciones cosmológicas. Para ello intentaremos optimizar el código y hacer uso del paralelismo intra-nodo mediante OpenMP, aprovechando la omnipresencia de procesadores *multicore* en las máquinas en las que se ejecuta.

1.2. Metodología y Planificación

Siendo conscientes de las dificultades y limitaciones que presenta la programación concurrente, deberemos inspeccionar la versión original del código para alcanzar un entendimiento suficiente de las estructuras de datos y algoritmos utilizados, y poder identificar así las zonas más adecuadas para aplicar la paralelización. Así mismo, analizaremos las distintas regiones de la aplicación para identificar aquellas zonas ya paralelizadas con MPI y las zonas en las que el código consume mayor tiempo y por tanto deberemos tratar de optimizarlas y paralelizarlas.

La metodología aplicada ha consistido en analizar la versión original haciendo uso de la documentación disponible, inspeccionando el código fuente y utilizando herramientas para el análisis de rendimiento como Gprof, Extrae y Paraver. Simultáneamente, con la ayuda del desarrollador principal, hemos analizado la estructura funcional de la aplicación y sus principales fases. Una vez identificada la región que más coste computacional representa, hemos diseñado e implementado un algoritmo alternativo buscando optimizar el rendimiento del código y que la paralelización posterior fuese lo menos compleja posible. Una vez implementada la versión alternativa secuencial de la región a optimizar, hemos analizado los tiempos de ejecución de cada una de las fases internas. Habiendo identificado la región que más tiempo consume, hemos procedido a paralelizarla con OpenMP. Por último, hemos realizado ejecuciones de casos de prueba para la versión original y para la nueva versión, comparando, tanto los resultados físicos obtenidos, como las diferencias en los tiempos de ejecución.

La planificación llevada a cabo podría sintetizarse de manera esquemática en los siguientes pasos:

1. Inspección de la versión original del código AHF, revisando su código fuente, realizando ejecuciones para familiarizarnos con la aplicación y analizando las distintas fases de ejecución.
2. Aplicación de herramientas de análisis de rendimiento para visualizar el comportamiento de la aplicación e identificar las regiones cuyo coste computacional sea mayor.
3. Diseño e implementación de las estructuras de datos y algoritmos que reemplazarán la región más costosa, en concreto aquella que realiza el refinamiento adaptativo de la caja y construye el árbol de patches.

4. Comprobación y comparación de los resultados obtenidos y los costes computacionales de la nueva versión frente a la versión original.

2. Herramientas para el análisis de rendimiento del software

Técnicas de profiling: muestreo, instrumentación, contadores hardware, generación de trazas y perfiles, visualización de resultados

El análisis de rendimiento del software, o perfilado (*profiling*), más aún en computación paralela, es una herramienta clave para entender cómo se comporta el código durante la ejecución. Nos permite detectar las "zonas calientes" (*hotspots* (37)) en las que se emplea más tiempo, descubrir cuáles son las rutinas que más veces se invocan, encontrar qué zonas pueden suponer un cuello de botella (*bottlenecks* (38)) serializando la ejecución y/o limitando el rendimiento del resto de la aplicación, conocer el comportamiento del hardware mediante la configuración y lectura de contadores (39), etcétera.

Todas estas técnicas de medida de rendimiento tienen la contrapartida inevitable de que afectan al comportamiento de la aplicación. Se deberá buscar la manera de utilizar la técnica más adecuada en función de la información en la que estemos interesados, teniendo siempre en cuenta que el propio análisis desvirtúa el funcionamiento natural de la aplicación inspeccionada (40).

2.1. Muestreo

Una de las técnicas más frecuentemente utilizadas es el muestreo (*sampling* (41)). Está basada en interrumpir periódicamente la ejecución de la aplicación que se desea analizar y consultar el estado de la aplicación (pila de llamadas, recursos utilizados, etcétera). La implementación más sencilla se basa en enviar señales de interrupción periódicamente a la aplicación y, en la función que captura la interrupción, ejecutar la recolección de información. Implementaciones más complejas pueden disparar el *sampling* en otros momentos basándose en parámetros de ejecución, como por ejemplo el valor de ciertos contadores hardware. Esta técnica tiene la ventaja de ser poco intrusiva en el comportamiento de la aplicación, ya que el código de la misma no se modifica. Sí se modifica el entorno de ejecución al cargar la función que captura la interrupción, por lo que se afectará, al menos, el comportamiento de las memorias cache.

Por otra parte, el resultado tiene limitaciones intrínsecas debido a su naturaleza estadística. Al capturar información con una periodicidad temporal, en distintas ejecuciones se puede interrumpir la ejecución en distintas zonas de la aplicación. El incremento de la frecuencia de muestreo podría dar resultados más próximos al comportamiento real de la aplicación, pero también aumentará el tiempo de ejecución del perfilado, la cantidad de información generada y el impacto del análisis en la ejecución respecto a la aplicación original.

2.2. Instrumentación

Otra de las técnicas más utilizadas es la instrumentación (42). En este caso, el código de la aplicación a analizar se modifica para poder recolectar información de ejecución en todas las zonas que se desee. Con esta técnica capturaremos todos los eventos que ocurran en las zonas que nos interesen, eliminando el error estadístico del muestreo. Como contrapartida tendremos una ejecución más desvirtuada frente a la aplicación analizada, ya que estaremos introduciendo instrucciones ajenas a la misma.

La forma más trivial de realizar la instrumentación de una función de usuario podría ser comparar el "tiempo de pared" (*wallclock*) antes y después de la llamada a la misma con funciones como omp_get_wclock() (43), MPI_Wtime() (44) u otras que proporcione el sistema operativo. Esta técnica, simple pero muy útil en muchos casos, introduce instrucciones en la aplicación, así como variables en las que almacenar las medidas de tiempo, por lo que estaremos modificando el comportamiento de la misma.

Métodos más complejos de instrumentación pueden sustituir las llamadas a funciones por otras rutinas que recolecten información sobre el estado de ejecución antes y después de llamar al código original de la función instrumentada. Esta es una de las técnicas más utilizadas por las herramientas de perfilado para aplicaciones paralelas, como TAU (45) y Extrae (46), mediante la instrumentación de las llamadas a las librerías de computación paralela más utilizadas: MPI, OpenMP, CUDA, OpenCL, etcétera. Estas librerías son sustituidas, en tiempo de compilación, o más frecuentemente en tiempo de enlazado dinámico previo a la ejecución, por otras librerías que capturan las llamadas a las librerías originales y, antes y después de llamar a las funciones originales, realizan la recolección de información pertinente.

Para ejecutar las llamadas de nuestra aplicación a las librerías instrumentadas, en vez de a las librerías por defecto, se suele utilizar una técnica basada en la variable de entorno LD_PRELOAD (47). El mecanismo de resolución simbólica mediante librerías dinámicas, al invocar una aplicación, comprueba las rutinas no definidas en el binario ejecutable y busca las mismas en librerías dinámicas en los directorios de librerías del sistema. Otra de las muchas variables de entorno que intervienen en este mecanismo es la variable de entorno LD_LIBRARY_PATH (47), la cual permite al usuario especificar directorios en los que el sistema de "linkado" buscará librerías para resolver los símbolos pendientes, antes de comprobar los directorios por defecto del sistema (48). La técnica basada en LD_PRELOAD utiliza el funcionamiento del "linkado" dinámico en tiempo de ejecución, de manera que, al cargar un binario ejecutable se cargarán en memoria también las librerías definidas en esta variable de entorno. Si estas librerías pre-cargadas contienen las funciones de las librerías de computación paralela que nuestra aplicación utiliza, se invocarán éstas en vez de las originales. Así se podrán capturar las llamadas a las librerías y a las funciones del *runtime* de los modelos de programación paralela que utilicemos, y podremos conocer qué proceso y/o *thread* invoca cada rutina paralela.

Además de este método, las herramientas de perfilado suelen proveer una API (49) (50) para que el programador pueda capturar información sobre el entorno de ejecución en los puntos que desee de su aplicación. Así se conseguiría instrumentar, además de las funciones de las librerías paralelas utilizadas, las funciones de usuario.

Para el análisis de rendimiento de la versión original (versión 1) como de la nueva (versión 2) del código AHF, se ha empleado el API de Extrae, en concreto la función "`void Extrae_user_function (int enter)`" que permite introducir un evento en la traza de ejecución correspondiente a la entrada (argumento enter 0) y salida (argumento enter 1) en una función de usuario. El evento insertado contiene información sobre el nombre del fichero fuente, el número de línea y el nombre de la función desde la que se invoca (51).

Ejemplo de uso de API de Extrae

Se han utilizado directivas del preprocesador de C para controlar la inclusión de llamadas al API mediante el macro EXTRAE_API_USAGE, que podrá definirse dentro del código y/o en tiempo de compilación

```
[…]
#ifdef EXTRAE_API_USAGE
#include <extrae_user_events.h>
#endif
[…]

void patches_generation(ahf2_patches_t** patches, subcube_t** sc_table, int32_t
initial_depth, int32_t final_depth, uint64_t NminPerHalo){

  /* Variables locales */

#ifdef EXTRAE_API_USAGE
  Extrae_user_function(1);
#endif

  /* Código de la función */

#ifdef EXTRAE_API_USAGE
  Extrae_user_function(0);
#endif

  /* return rc en funciones cuyo tipo sea distinto de void */
}
```

Junto con la interfaz de programación explicada anteriormente, existe la posibilidad de modificar los ficheros binarios ejecutables de una aplicación, mediante la utilización del proyecto Paradyn (52) y su API Dyninst (53). Tanto TAU (54) como Extrae (55) permiten utilizar Dyninst para instrumentar automáticamente y en tiempo de ejecución las funciones de usuario deseadas. Al instrumentar funciones de usuario debemos tener en cuenta las implicaciones que esto puede tener: si capturamos la ejecución de funciones llamadas un gran número de veces dentro de bucles, o debido a las llamadas recursivas de las mismas, el *overhead* en el tiempo de ejecución y el volumen de información generada pueden ser prohibitivos.

2.3. Contadores hardware

Por otra parte, los microprocesadores modernos implementan unos contadores de rendimiento hardware que combinan un limitado conjunto de registros junto con la capacidad de acceder a información de comportamiento del hardware. El ejemplo de uso más básico de estos contadores sería la capacidad de acceder al contador de ciclos de reloj así como al contador de instrucciones completadas por el procesador, pudiendo calcular el CPI (*Cycles per*

Instruction) en distintas regiones de nuestra aplicación. Hoy en día son muchos los contadores hardware disponibles en los procesadores modernos, permitiendo contabilizar los fallos de cache (*cache misses*) en los distintos niveles L1, L2 y L3, para cache de instrucciones, de datos y el total de ambas. Existen también contadores hardware que permiten inspeccionar el comportamiento de las unidades de coma flotante, los aciertos y fallos en las predicciones de saltos, contar los fallos y aciertos (*misses* y *hits*) en el acceso al TLB (56), contar el número de instrucciones que hacen uso de unidades vectoriales/SIMD, etcétera. (57)

El acceso a estos contadores para los programadores se suele realizar mediante el proyecto PAPI (Performance Application Programming Interface) (58), un API que provee acceso a los contadores hardware disponibles en el microprocesador y permite configurar los contadores que se quieren leer y almacenarlos en los registros disponibles.

2.4. Perfiles y trazas

Estas técnicas, junto a las herramientas complementarias como la instrumentación dinámica o el *sampling* de contadores hardware, pueden generar una gran cantidad de información sobre la ejecución de la aplicación que queremos analizar.

Existen dos posibles resultados del perfilado de una aplicación. El perfil (*profile*) es el procesado estadístico de la información recolectada durante la ejecución que muestra las funciones que se han invocado y qué porcentaje de tiempo han utilizado cada una de ellas, durante la ejecución de su código propio y el total acumulado de su código junto con el tiempo empleado en llamadas a subrutinas. Este resultado recibe el nombre de perfil plano (*flat profile*). Esta información puede combinarse con detalles sobre qué funciones llaman a otras funciones, generando lo que se conoce como perfil de grafo de llamadas (*call graph profile*), el cual permite inspeccionar con más detalle las relaciones entre las distintas funciones.

Por otra parte, el conjunto de información recolectada, sin realizar ningún procesado estadístico sobre la misma, da una visión de los eventos que han ocurrido a lo largo de toda la ejecución, ya sean eventos generados mediante muestreo o eventos generados por código de instrumentación insertado en la aplicación. Toda esta información genera lo que se conoce como trazas de ejecución, normalmente una por cada uno de los procesos y *threads* de ejecución, que debe combinarse para obtener uno o varios ficheros que aglutinen la información generada por toda la aplicación.

2.5. Formatos de trazas

En el caso de Extrae, los ficheros de trazas generados tienen un formato propio con extensión .mpit, .mpits y .sym. En el paquete de código se incluyen herramientas para la conversión de estos ficheros en un formato compatible con el software Paraver (59), la herramienta de visualización de trazas de ejecución compatible con Extrae.

En el caso de TAU son varios los formatos en los que puede generar las trazas. La herramienta de visualización proporcionada es Paraprof (60) y es capaz de leer gran número de formatos (61) además de los formatos propios que TAU genera.

Dada la dificultad y complejidad que puede suponer trabajar con distintos formatos de trazas de ejecución al analizar el rendimiento de una aplicación y la obligatoriedad de utilizar las herramientas de visualización compatibles, surgió el proyecto OTF (*Open Trace Format*) (62) que provee un formato para las trazas así como una librería para acceder a las mismas tratando de estandarizar un formato común para todas las herramientas que generen o visualicen trazas. Una evolución de este proyecto es OTF2 (63) (64) (65), que se convertirá en el formato estándar para herramientas de análisis de rendimiento como Scalasca (66), Vampir (67) y TAU.

2.6. Gprof

Cabe también mencionar la herramienta Gprof (68) (69). Su funcionamiento se basa en una combinación de *sampling* e instrumentación. El código de instrumentación se inserta en tiempo de compilación, utilizando el flag –pg de GCC. La ejecución de la aplicación generará unas trazas en uno o varios ficheros con extensión .gmon. La lectura de estos ficheros con el comando Gprof muestra un perfil plano que indica el tiempo de ejecución de cada función de usuario, el número de llamadas que se han realizado, así como el porcentaje de tiempo de ejecución propio de la función y el porcentaje acumulado considerando la ejecución de la función en sí y las llamadas a otras rutinas desde la misma. Por otra parte, también se muestra un perfil de grafo de llamadas con información sobre desde qué funciones se invoca cada función y a qué funciones hijas llama cada una, así como información de invocaciones recursivas.

Para la realización de este Trabajo Final de Máster se han utilizado las herramientas Extrae (haciendo uso del API de Extrae, de la instrumentación dinámica de Dyninst y de contadores hardware) y Paraver para el análisis de rendimiento de las ejecuciones paralelas de la versión 1 y 2 de AHF. También se ha utilizado Gprof durante el desarrollo de la versión secuencial de la versión 2 de AHF. En los siguientes capítulos se mostrarán detalles de la utilización de estas herramientas y análisis de los resultados obtenidos.

2.7. Herramientas propias de Intel y AMD

Otras herramientas muy útiles son las que los propios fabricantes de hardware proporcionan. Tanto Intel como AMD, como principales fabricantes de microprocesadores para ordenadores personales y servidores, han desarrollado sus propias herramientas de desarrollo de software (SDK, *Software Development Kit*). Éstas suelen incluir compiladores de C, C++ y Fortran, y librerías de cálculo científico optimizadas para sus propias arquitecturas que usualmente implementan, al menos, las librerías algebraicas BLAS y LAPACK, la transformada rápida de Fourier (FFT) y la generación de números aleatorios. Quizá el conjunto de herramientas de este

tipo más ampliamente conocido sea el proporcionado por Intel, principalmente el conjunto de compiladores icc, para C y C++, e ifort, para Fortran, junto con las librerías MKL (*Math Kernel Library*). AMD por su parte también proporciona el compilador x86Open64 para C, C++ y Fortran y la librería científica ACML (*AMD Core Math Library*), aunque parece no haber alcanzado ni el rendimiento ni la popularidad que las correspondientes de Intel.

Junto con lo anterior, suelen incluir herramientas para analizar el rendimiento de aplicaciones y generar un perfilado y trazado de las mismas. Estas herramientas están diseñadas para que los desarrolladores obtengan una visión global de las zonas de sus aplicaciones en las que la ejecución consume más tiempo, así como identificar posibles cuellos de botella en la paralelización, tanto en memoria compartida como distribuida, pero se asume un conocimiento más superficial de los detalles sobre perfilado de aplicaciones por parte del usuario. Además suelen utilizar interfaces gráficas que facilitan enormemente su utilización, pero sobrecargan las necesidades de memoria y cómputo durante el perfilado. En concreto, la herramienta VTune Amplifier XE, de Intel, permite generar perfiles de ejecución detallados con información sobre la pila de llamadas de la aplicación, el comportamiento de los cerrojos y las esperas de los distintos hilos de ejecución, las invocaciones de rutinas OpenCL en aceleradores, la rápida identificación de zonas calientes (*hotspots*) en el código, y un largo etcétera. Además, al ser herramientas desarrolladas por el fabricante del hardware, permite analizar los detalles internos del comportamiento del procesador en que se ejecuta. Se intuye que en el caso de utilizar procesadores de AMD, será mejor utilizar las herramientas proporcionadas por este fabricante. A pesar de lo anterior, su utilización es posible en procesadores diferentes al propio del fabricante del hardware y la herramienta, aunque el análisis será menos preciso en lo concerniente a los detalles internos del microprocesador.

Como contrapartida, una vez superada la curva de aprendizaje de estas herramientas de análisis de rendimiento, podría suponer una dificultad añadida tener que aprender a utilizar la desarrollada por otro fabricante cuando la máquina en la que se ejecute un código sea diferente. Por este motivo hemos optado, durante este trabajo, por utilizar herramientas independientes del fabricante.

3. AHF versión 1

Situación de la versión actual de AHF

El código AHF es el resultado de más de 15 años de desarrollo de códigos de simulaciones y de detección de halos. Los orígenes de este software se remontan a 1997 con la creación de MLAPM (*Multi-Level-Adaptive-Particle-Mesh*) que implementaba una técnica de subdivisión regular de la caja (*grid*). La primera publicación de este código data de 2001. En 2004 se crea el código MHF (*MLAPM Halo Finder*), que realiza el reconocimiento de halos *on-the-fly* integrado con el código de simulación. El siguiente año, en 2005, se renombran los proyectos pasando MLAPM a llamarse AMIGA (*Adaptive Mesh Investigations of Galaxy Assemby*) y MHF se renombra a AHF (*AMIGA Halo Finder*). El proyecto continúa evolucionando y en 2007 se publica la primera versión paralelizada con MPI. En 2012 se separa el código de simulación (AMIGA) del código de búsqueda de halos (AHF) (36) (70).

Para la redacción de este capítulo se han utilizado los siguientes artículos como referencia:

- Gill S.P.D., Knebe A., Gibson B.K., 2004, MNRAS, 351, 399. (71)
- Knollmann S.R., Knebe A., 2009, ApJS, 182,608 (72)
- Knebe A., AHF-AMIGA's Halo Finder, User's Guide (70)
- Knebe A., Green A., Binney J., MNRAS (2001) 325 (2): 845-864. (73)
- Alexander Knebe et al., 2013MNRAS.435.1618K. (74) (75)

3.1. Estructura de la aplicación

La aplicación tiene ciertas regiones claramente diferenciadas que podríamos enumerar de la siguiente manera:

1. El primer paso consiste en leer el fichero de configuración en el que se especifican los parámetros de la ejecución, así como los ficheros de input que contienen las partículas de la captura (*snapshot*) de la simulación a analizar. Este fichero de configuración se pasa como argumento al ejecutar la aplicación.

2. Se procede con la lectura de los ficheros de input. En el caso de que sea una ejecución paralelizada con MPI, cada tarea MPI leerá en paralelo el número de partículas que le corresponda procesar.

3. Para cada partícula se calcula su índice correspondiente a la *space-filling curve* (SFC en adelante) basada en la curva de Hilbert (76) (ver sección 3.2.1). Siguiendo el orden de los índices de esta SFC, se procede a realizar la descomposición de dominio repartiendo bloques de partículas de tamaño similar a cada tarea MPI. Así se

conseguirá enviar conjuntos de partículas que mantengan la localidad y a la vez el número de partículas asignadas a cada tarea será equivalente. Una descomposición de dominio que no utilizase una SFC podría optar por distribuir divisiones estáticas de la caja en la que no se tendría en cuenta el balanceo de carga en el probable caso de que el número de partículas en cada división de la misma no fuese equivalente. Por otra parte, si se optase por garantizar que cada tarea recibiese el mismo número de partículas ignorando la posición de las mismas, las comunicaciones necesarias posteriormente entre las tareas MPI serían muy frecuentes y costosas, ya que cada tarea tendría que comprobar si alguna otra tiene partículas en la misma región que esté analizando.

Junto con el conjunto de partículas que corresponden a cada tarea se envían también aquellas partículas adyacentes al volumen que le corresponde. La selección de estas partículas, que formarán un buffer para evitar comunicaciones posteriores entre tareas, se basará en las partículas cuya clave SFC se corresponda con las posiciones contiguas al volumen adjudicado a cada tarea. Es importante recordar que se imponen condiciones de periodicidad en la caja. Con este buffer de partículas se producirá replicación de partículas entre tareas, pudiendo encontrarse un mismo halo en dos tareas distintas. Para evitar esto, cada tarea solamente considerará los halos cuyo centro se encuentre en la región de la caja que estrictamente le corresponda. Aunque las partículas del buffer no puedan generar halos en una tarea, serán necesarias para calcular propiedades físicas de los halos de sus regiones.

4. En este punto de la ejecución cada tarea MPI tiene el conjunto de partículas que debe procesar. En el caso de una ejecución que no utilice MPI, el proceso único tendría en memoria todas las partículas de la captura a analizar. El siguiente paso que se lleva a cabo es aplicar una división regular de la caja, generando una malla del tamaño configurado por el usuario a través del fichero de configuración, en el parámetro *DomGrid*. Al trabajar en 3 dimensiones, la división de la caja dará lugar a $DomGrid^3$ celdas. Para cada una de las celdas de este mallado se calcula la densidad de partículas correspondiente, es decir, el número de partículas que se encuentran dentro de la misma.

5. Una vez realizado el mallado de la caja y la asignación de partículas a cada celda, se aplica un método de refinamiento adaptativo. Para ello se comprobará si las celdas contienen un número de partículas por encima de un umbral, también definido por el usuario como parámetro. En el caso de que la densidad de una celda sobrepase el umbral, se procederá a subdividirla en el siguiente nivel de refinamiento, y asignar a cada una de las nuevas celdas las partículas que le correspondan. La división de una celda por la mitad, para alcanzar el siguiente nivel de refinamiento, dará lugar a 8 (2^3) nuevas celdas al realizarse la división en las 3 dimensiones del espacio. Si las nuevas celdas siguen superando el umbral de número de partículas por celda se continuará refinando hasta que ninguna celda necesite ser dividida. Aquellas celdas que resulten vacías al no corresponderles ninguna partícula no se crearán.

6. Una vez que haya finalizado el proceso de refinamiento, obtendremos una jerarquía de refinamiento que permitirá reconocer las zonas en las que se concentran el mayor número de partículas o, utilizando terminología astrofísica, trazará el campo de densidad, lo cual podrá ser utilizado para identificar estructuras cosmológicas.

7. Los conjuntos de celdas adyacentes que formen bloques aislados, en cada uno de los niveles de refinamiento, deberán ser detectados y almacenados como posibles halos. El término utilizado para denominar estos bloques aislados de celdas o nodos adyacentes es *patch* (del inglés: parche, zona, área). Cabe resaltar que un patch, en un determinado nivel de refinamiento, puede haber formado, en niveles de refinamiento más fino, sus propios patchs (*daughter patchs*). De manera recíproca, cualquier patch se corresponderá con un único patch en el nivel de refinamiento consecutivo y más grueso (*parent patch*). Esto ocurre de manera natural al aplicar el algoritmo de refinamiento adaptativo y será la clave para encontrar sub-estructuras dentro de otras estructuras. Esta propiedad del algoritmo tiene una profundidad solamente limitada por el nivel más grueso de refinamiento (el grid inicial), definido por el parámetro *DomGrid*, y por el nivel más fino de refinamiento, que estará determinado por el umbral que defina el número de partículas máximo que puede tener un nodo sin que deba ser refinado, así como por el número y la distribución de las partículas en la captura de la simulación.

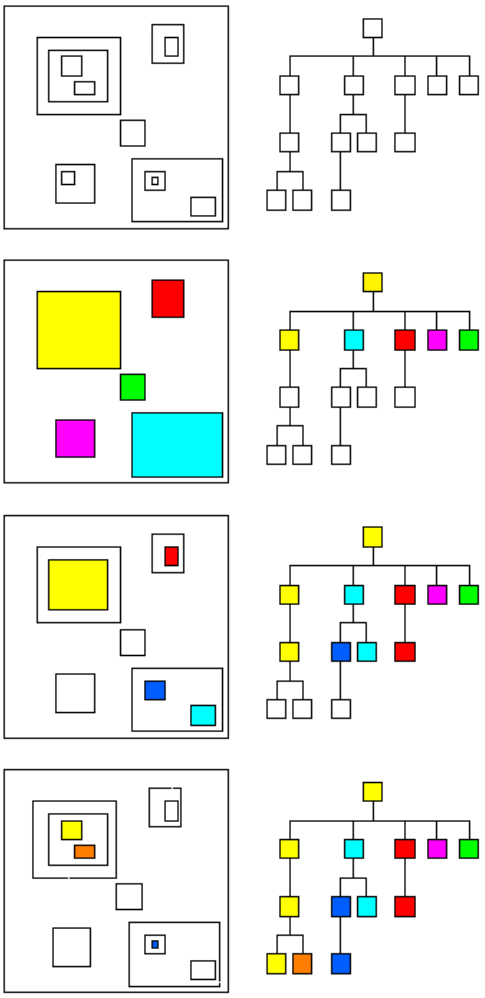

Figura 6. “process of classifying a grid tree into substructures” (72)

8. Llegados a este punto contaremos con un conjunto de patches organizados en forma de árbol (ver Figura 6), o muy probablemente de bosque (conjunto de árboles). Cada una de las hojas de estos árboles será un candidato para convertirse en un halo.
   Para cada uno de los candidatos a halo se contabilizarán las partículas incluidas en su patch y se calcularán propiedades físicas del mismo como el potencial gravitatorio, el centro de masa, la velocidad de escape correspondiente a cada partícula, etcétera. En

este momento se aplica un procedimiento iterativo para descubrir si una partícula perteneciente a un halo está o no ligada gravitacionalmente. En el caso de que no esté ligada, no se tendrá en cuenta su participación para las propiedades del halo, pero sí se considerará para niveles superiores (con grados de refinamiento más gruesos). Con las nuevas propiedades del posible halo, vuelve a considerarse si las partículas están ligadas gravitacionalmente o no, hasta que se compruebe que todas las partículas consideradas parte del halo están efectivamente ligadas al mismo.

Una vez encontrados varios halos en un determinado nivel, se procederá a comprobar su fusión en el nivel inmediatamente superior. En este momento debe tomarse la decisión, de gran relevancia cosmológica, de cuál es la rama principal (*trunk branch*) en el árbol, escogiendo como tal la que contenga el halo con mayor número de partículas. El procesamiento de la jerarquía de patches basado en los parámetros y criterios cosmológicos dará lugar a una jerarquía de halos, en la que se formarán halos padres (*host halos*) y subestructuras como subhalos y sub-subhalos.

9. Por último se generarán los ficheros de output que son de cuatro tipos. El fichero de halos consistirá en un catálogo de los halos encontrados en la captura de la simulación. El fichero con los perfiles de cada uno de los halos, que contendrá los valores de densidad, potencial gravitacional y otros muchos parámetros físicos, a diferentes radios esféricos con centro en el centro del halo. Otro fichero con la jerarquía de subestructuras listará todos los sub-halos que contiene cada halo. También se genera un fichero de todas las partículas que estén ligadas a alguno de los halos encontrados, con sus propiedades de posición, velocidad, etcétera

3.2. Conceptos clave de la implementación

Existen un conjunto de características en la implementación que merecen una mención especial.

3.2.1. Space-filling curve

Este tipo de curvas SFC citadas anteriormente, fueron inicialmente desarrolladas por Giuseppe Peano a finales del siglo XIX (77). David Hilbert posteriormente extendió el trabajo sobre este tipo de curvas.

La curva de Hilbert consiste en un mapeo entre las coordenadas (*x,y,z*) de una posición en el interior de la caja a un identificador *d* unidimensional (ver Figura 7). Estos identificadores tiene la propiedad de implementar un recorrido ordenado y continuo en un espacio –normalmente en 2D y 3D, pero ampliable a N dimensiones- que mantiene propiedades de localidad. La curva garantiza que el mapeo a coordenadas de identificadores consecutivos dará posiciones próximas. El razonamiento inverso, en cambio, no puede cumplirse siempre, ya que en el espacio tridimensional el número de coordenadas adyacentes es mayor que en el espacio unidimensional de la curva.

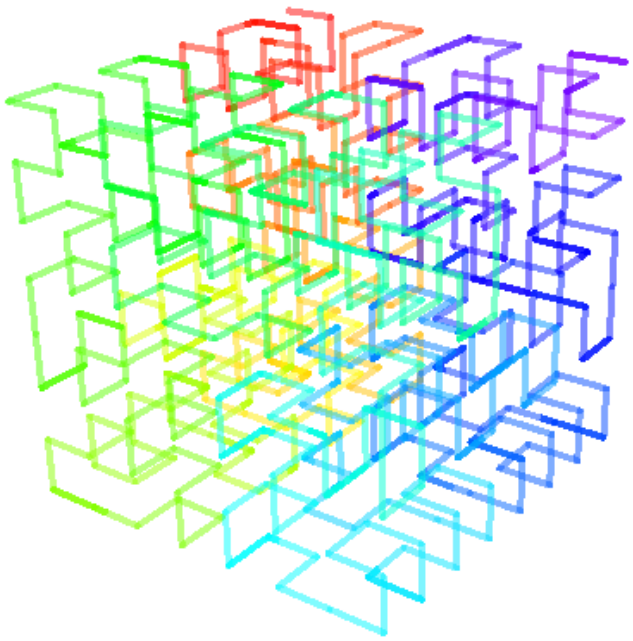

Figura 7. 3-D Hilbert curve with color showing progression. Wikipedia.org.
Autor: Robert Dickau. Licencia CC BY-SA 3.0.

La precisión de la división de la caja por la curva SFC vendrá dada por el parámetro de ejecución *LevelDomainDecomp* (*LDD*), proporcionado por el usuario mediante el fichero de configuración. Como norma general, cada dimensión del espacio a ser recorrido tendrá 2^{LDD} divisiones. Como se puede ver en el siguiente extracto del log de ejecución, para un valor de LDD=6, se realizan 2^6 divisiones por dimensión, $(2^6)^3 = 262144$.

Fragmento del log de ejecución generado por AHF

```
[…]

Setting volume boundaries
  minkey: 0
  maxkey: 262143
  level : 6
  ctype : Hilbert curve

[…]
```

3.2.2. Normalización de coordenadas

Con un propósito de simplicidad y de abstracción frente a las características particulares de la caja de la simulación, durante el post-procesado se asume que el tamaño de la caja es 1. Por tanto, es necesario escalar todas las coordenadas de las partículas y normalizarlas entre 0 y 1. Más adelante veremos cómo esta característica ha facilitado la implementación de la *cubekey* en la nueva versión.

3.2.3. Descomposición de dominio, reparto de carga MPI y limitaciones

Como se ha explicado anteriormente, la descomposición de dominio y el reparto de la carga computacional entre cada tarea MPI se realiza mediante la asignación de un número

equivalente de partículas a cada una. El orden de distribución de partículas sigue la SFC para mantener la localidad dentro de lo posible. Aún así, no se garantiza que esta distribución de la carga preserve la asignación de todas las partículas que componen una estructura cosmológica (halo o subhalo) a una única tarea MPI. La implementación no realiza ninguna comunicación MPI posterior, por lo que la división de las partículas de una estructura entre varias tareas daría resultados erróneos, generando varios halos de menor tamaño, en vez del halo mayor que correspondería con el resultado correcto. Este efecto se aminora con el envío del buffer de partículas que, aunque no correspondan estrictamente a la tarea, se consideran para el cálculo de propiedades físicas de los halos. Por otra parte, la selección de las partículas que formarán parte de este buffer se realiza mediante la búsqueda de todas las partículas adyacentes al dominio asignado, basándose en el valor de la SFC. Es decir, para cada valor de la SFC de las partículas en el contorno del volumen asignado, se comprobará cuáles no están incluidas en el mismo pero son contiguas. Esos valores de la SFC designarán qué partículas formarán el buffer. Por tanto, cuanto mayor sea el nivel de descomposición de dominio utilizado para el cálculo de la SFC, menor tamaño tendrán las subdivisiones y menor será el buffer de partículas que evitaría generar resultados erróneos.

Otros parámetros que podrán influir en el efecto de la descomposición de dominio sobre los resultados será el momento cosmológico con el que se corresponda la simulación que se esté analizando. En el origen del universo no existían estructuras y había algunas perturbaciones o irregularidades (anisotropías) en la distribución homogénea de materia y energía. Con el paso del tiempo comenzaron a formarse estructuras como galaxias y cúmulos de galaxias, y posteriormente estos se fueron fusionando. La distribución enormemente heterogénea de objetos en etapas avanzadas de la historia del universo provoca que la probabilidad de encontrar objetos en una región aleatoria sea muy baja. Parece por tanto predecible la mayor probabilidad de dividir una estructura entre dos tareas MPI cuanto mayor sea la antigüedad de la simulación, en la escala de tiempo cosmológica, la cual se suele medir como valor del *redshift*.

Por otra parte, es frecuente realizar re-simulaciones de ciertas regiones de simulaciones anteriores. Así, las zonas en las que se han generado estructuras relevantes en una simulación, se pueden mapear a las regiones de las condiciones iniciales de las que proceden y re-simular solamente estas regiones con una mayor resolución mediante la utilización de un mayor número de partículas de menor tamaño en una caja más pequeña. En estos casos, es probable que las estructuras relevantes ocupen grandes áreas de la caja, por lo que aumentarán las probabilidades de dividir las partículas que forman una estructura entre diferentes tareas MPI.

Por último, cabe resaltar que los inconvenientes citados anteriormente también se harán más difícilmente evitables según aumente el número de tareas MPI, lo cual limita claramente la escalabilidad de la implementación.

3.3. Estructuras de datos

En todo desarrollo de software, las estructuras de datos utilizadas suponen un factor clave que afectará a los algoritmos que se podrán aplicar y al rendimiento de los mismos. Cuando se afronta el desafío de diseñar las estructuras de datos que utilizaremos para manejar las partículas de simulaciones cosmológicas debemos tener en cuenta varios aspectos.

Por un lado, el número de partículas que manejaremos se encuentra en el rango de los miles de millones. Solamente para almacenar el identificador de cada una de las partículas deberemos considerar la utilización de enteros de 64 bits, ya que el límite de un entero sin signo de 32 bits (2^{32} = 4.294.967.296) podría alcanzarse fácilmente. Ya vimos en la introducción que se han realizado simulaciones de más de cincuenta mil millones de partículas. Deberemos optar por tanto por identificadores de 64 bits para cada partícula. Junto con el identificador deberemos almacenar, al menos, 3 valores decimales para las coordenadas de la posición y otros 3 para la descomposición en los 3 ejes del espacio del vector de velocidad. Dependiendo de si los valores reales los almacenamos como variable en coma flotante de simple o doble precisión, necesitaremos 4 u 8 bytes para cada valor. Un cálculo rápido nos podría indicar que para almacenar la información mínima de cada partícula necesitaremos, al menos, 8 bytes para el ID y 4 u 8 bytes para cada uno de los 6 valores almacenados en coma flotante, lo que nos dará un tamaño de entre 32 y 56 bytes por partícula. Teniendo en cuenta que un Gigabyte de memoria equivale aproximadamente a mil millones de bytes, podemos deducir que para cada mil millones de partículas necesitaremos entre 32 y 56 GBytes de memoria, sin tener en cuenta los posibles costes de memoria adicional que puede suponer la propia estructura de datos si utilizamos algo más sofisticado y usable que un simple array. Estas inmensas necesidades de memoria nos indican claramente dos aspectos a tener en cuenta. El primero: hoy en día es difícil encontrar máquinas o nodos de supercomputadores con 50 GBytes de memoria, por lo que la paralelización utilizando el modelo de memoria distribuida (MPI principalmente) va a ser inevitable. El segundo aspecto que debemos considerar es la gran importancia de minimizar al máximo el tamaño de las estructuras de datos que vayamos a utilizar. Inspeccionando logs de ejecución podemos afirmar que el refinamiento adaptativo de la caja generará un número de celdas similar al número de partículas de la simulación. Por tanto, no sólo debemos minimizar el tamaño de la estructura empleada para almacenar las partículas, sino también el de aquellas que sirvan para representar cada celda de las subdivisiones.

Aún sin conocer los detalles de implementación, gracias a la colaboración con su desarrollador principal y co-tutor de este trabajo, Alexander Knebe, junto con la información publicada en el artículo Knebe A., Green A., Binney J., MNRAS (2001) (73), intentaremos hacer una descripción aproximada de la situación de la versión 1 de AHF.

Según se aplica la división inicial de la malla de nivel más grueso, cada una de las celdas creadas que contengan partículas dará lugar a una estructura llamada NODE que, junto con propiedades físicas de la misma, almacena un puntero a la primera partícula que pertenece a la celda. Cada partícula tiene su propia estructura, con la cual se forma un array de estructuras. Este array de partículas será ordenado en función del valor que le corresponda en la SFC, pero cada estructura de partícula tendrá a su vez un puntero a la siguiente partícula, formando así una lista enlazada de partículas. El orden en esta lista enlazada no se corresponde con el índice SFC, sino con su posición en el eje de coordenadas x. Por último, existe una estructura de

mayor nivel llamada QUAD (tres estructuras realmente: xQUAD, yQUAD y zQUAD) que almacenará los grupos de celdas adyacentes en cada uno de los ejes. La estructura xQUAD está compuesta por un puntero al primer NODE, su posición en el eje x, su longitud –número de celdas contiguas-, y el puntero al siguiente xQUAD que contendrá el siguiente grupo de celdas. De manera análoga, las estructuras yQUAD contendrán la información sobre los grupos de xQUAD, y a su vez las estructuras zQUAD proveerán acceso a las estructuras yQUAD (ver Figura 8).

Para que estas estructuras de datos funcionen se deben cumplir ciertos requisitos, como el orden en las listas enlazadas de las partículas y la asignación de memoria en bloques contiguos para las estructuras NODE, lo que permite recorrerlas, dentro de un QUAD, incrementando el puntero.

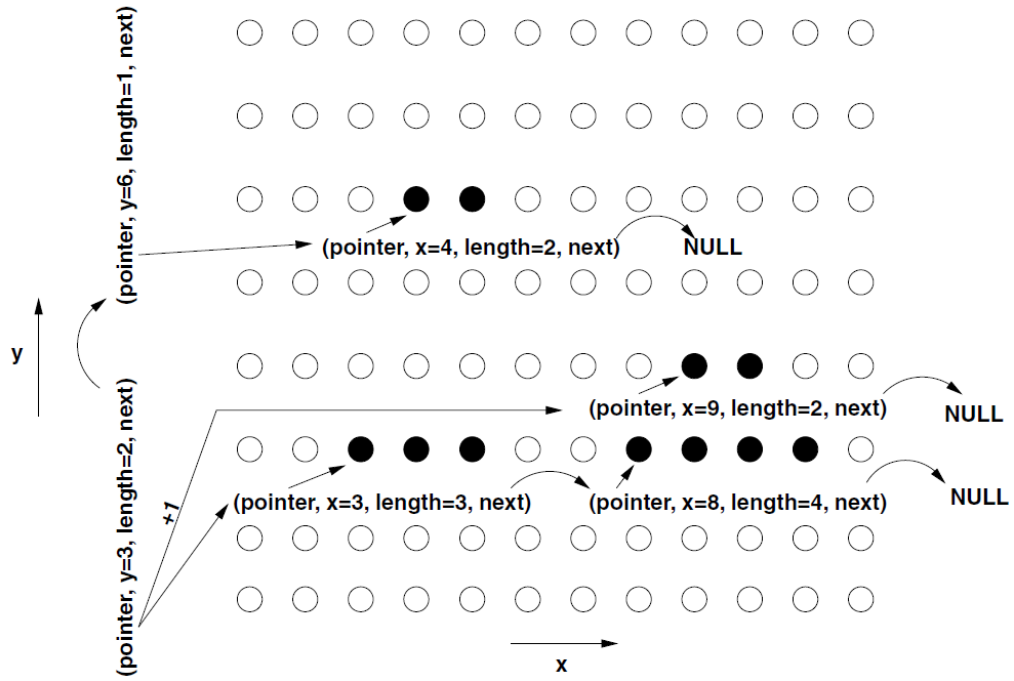

Figura 8. "QUAD structured grid […] sketched for 2 dimensions. […] QUADs are indicated by lists in brackets". Credit: Imagen extraída del artículo Knebe A., Green A., Binney J., MNRAS (2001) (73)

3.4. Perfilado de la aplicación

Para proceder con el análisis de rendimiento y perfilado de la versión 1 de AHF hemos utilizado las herramientas Extrae para generar las trazas de ejecución y Paraver para visualizarlas y obtener una representación gráfica del comportamiento de la aplicación.

3.4.1. Simulaciones analizadas

La comunidad científica dedicada al desarrollo de herramientas para el post-procesado de simulaciones cosmológicas, así como al desarrollo de códigos de simulaciones de N-cuerpos, ha realizado varios encuentros en los últimos años para la comparación de los resultados de estas

herramientas, con el objetivo de aunar esfuerzos y criterios para la obtención de resultados compatibles. El primero, *Haloes Going MAD*, tuvo lugar en Miraflores de la Sierra, Madrid, en el año 2010 (78). Desde entonces se han producido varios encuentros más, como el *Subhaloes going Notts* cerca de Nottingam el año 2011 (79). El último ha tenido lugar en el Instituto de Física Teórica de la Universidad Autónoma de Madrid, durante el mes de Julio de 2014, con una duración de 3 semanas (80).

Como *input* hemos utilizado dos de los *tests* aceptados comúnmente a raíz de estos encuentros (81). Uno de ellos, procedente del encuentro *Haloes Going MAD* consiste en varias simulaciones de una caja con 64^3, 128^3, 256^3, 512^3 y 1024^3 partículas. Estos inputs, según las indicaciones del desarrollador principal de AHF, son los únicos que pueden utilizarse con la versión MPI del código, dadas las limitaciones de la descomposición de dominio explicadas anteriormente (el resto de casos de input son re-simulaciones de alta resolución). En concreto, hemos utilizado la simulación de 64^3 (262.144) partículas para pruebas en la máquina de trabajo personal y las simulaciones de 128^3 (2.097.152), 256^3 (16.777.216) y 512^3 (134.217.728) para comprobar el comportamiento de la paralelización con MPI y OpenMP.

El otro test utilizado es una simulación resultante del proyecto CLUES (*Constrained Local UniversE Simulations*) (82), dedicado a simular el Grupo Local (83), que es el conjunto de galaxias en que nos encontramos, formado por la Vía Láctea, Andrómeda y la Galaxia del Triángulo, junto con un grupo de galaxias menores. La relevancia de simular esta región cercana se debe a que es, evidentemente, la mejor observada del universo. Este test está formado por 120 millones de partículas, pero no puede ejecutarse con la versión paralela MPI debido a las limitaciones de la descomposición de dominio ya citadas.

3.4.2 Máquinas utilizadas

Para la realización de los análisis de rendimiento de la aplicación hemos utilizado dos entornos diferentes:

- **Kahan cluster**. Este clúster de computadores, perteneciente al DSIC de la UPV, está formado por 6 nodos, siendo cada uno un bi-procesador con un total de 32 cores y 32 GB de memoria RAM (84). La ejecución de trabajos en el cluster se realiza mediante el sistema de colas PBS-Torque.

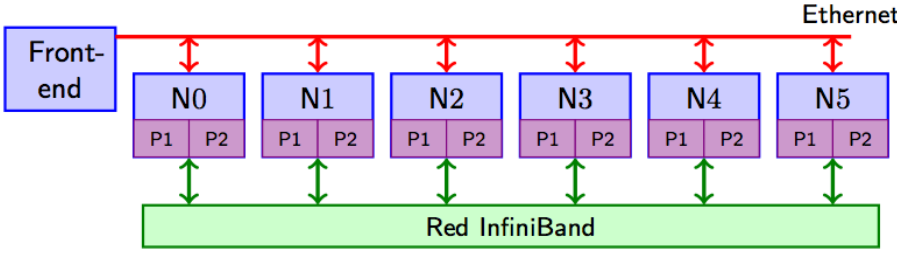

Figura 9. Diagrama del clúster Kahan (84)

Figura 10. Visualización de la topología del nodo kahan02 generada con *lstopo*.

- **Brutus.ft.uam.es**. Esta máquina, perteneciente al Grupo de Astrofísica y Cosmología del Departamento de Física Teórica de la Universidad Autónoma de Madrid (85), cuenta con un total de 64 cores distribuidos en 4 procesadores de 16 núcleos cada uno y con 1TB de memoria RAM. Utiliza Slurm como sistema de colas.

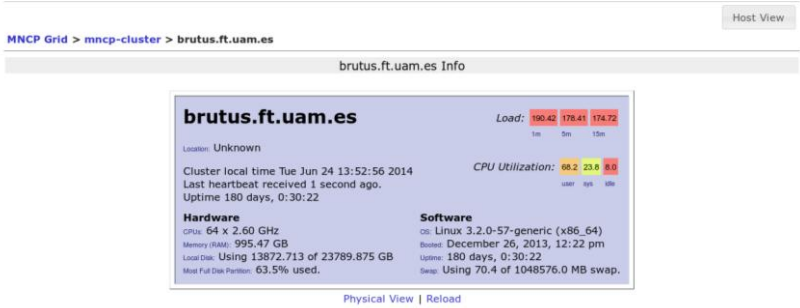

Figura 11. Visualización de la máquina brutus mediante el sistema Ganglia

Debido a la gran cantidad de memoria requerida para trabajar con simulaciones cosmológicas, más la sobrecarga en tiempo de ejecución y requisitos de memoria que supone utilizar una herramienta de perfilado, para ciertos casos hemos tenido que recurrir a la máquina brutus.

Figura 12. Visualización de la topología de la máquina brutus.ft.uam.es generada con *lstopo*.

3.4.3. Instalación de herramientas de perfilado

Respecto a la utilización de la herramienta de generación de trazas de ejecución Extrae, debemos resaltar que su instalación y utilización no es trivial. Las posibilidades de descarga que los desarrolladores ofrecen son básicamente dos: formato binario/ejecutable y código fuente. Optar por la versión binaria puede parecer *a priori* la opción más sencilla, pero al tratarse de un directorio con los ejecutables y librerías propia, en vez de un paquete adaptado al gestor de paquetes de la distribución GNU/Linux que utilicemos, es frecuente encontrar dependencias *rotas* de las librerías externas de las que hace uso la aplicación, así como referencias a directorios de estas librerías con configuraciones distintas a las de nuestras máquinas. Algunas de las dependencias son la librería libxml2, utilizada para leer el fichero de parámetros que Extrae necesita; libunwind, utilizada para obtener la pila de llamadas de una aplicación en ejecución; PAPI, interfaz de programación mencionado anteriormente que permite acceder a los contadores hardware del microprocesador; DynInst, librería que permite la instrumentación de funciones de usuario en tiempo de ejecución; libbfd o libelf, que permiten acceder y manipular los ficheros ejecutables en formato ELF, estándar en GNU/Linux. Por supuesto, también es necesario tener instalado el entorno de ejecución de MPI y OpenMP, utilizado por nuestra aplicación.

Debido a los errores obtenidos al intentar ejecutar la versión ejecutable proporcionada, tuvimos que optar por la compilación e instalación desde el código fuente.

3.4.4. Fichero de configuración de Extrae (extrae.xml)

Una vez instalada la herramienta de generación de trazas, tuvimos que adaptar fichero de configuración para capturar los parámetros de perfilado que nos interesaban (86). Este fichero, en formato XML, llamado por defecto extrae.xml, permite, en la sección inicial, habilitar la generación de trazas, indicar el directorio de instalación de Extrae en el sistema y especificar el tipo de trazas que deseamos (Paraver en nuestro caso, ya que utilizaremos esta herramienta posteriormente para visualizar el análisis). Las siguientes secciones permiten activar la recolección de información sobre llamadas MPI y OpenMP. Otra sección relevante, llamada *Callers*, permite obtener la pila de llamadas que efectúan las invocaciones a rutinas MPI y OpenMP. La sección *user-functions* es donde definiremos las funciones de usuario que queremos instrumentar mediante el uso de la librería Dyninst. Uno de los atributos de esta sección, llamado *list*, se usa para proporcionar el *path* absoluto al fichero en el que listaremos las funciones de usuario a instrumentar. Otra sección fundamental es la llamada *counters* que permite activar la recolección de contadores hardware mediante PAPI, así como contadores hardware de los interfaces de red Myrinet (anteriormente usados en el supercomputador Marenostrum, pero prácticamente en desuso hoy día), contadores del sistema operativo que hacen uso de la llamada al sistema *getrusage* (87). Las dos últimas secciones del fichero XML que hemos utilizado son la que hace referencia al muestreo periódico de la aplicación (*sampling*) y la que hace referencia a la fusión (*merge*) automática de las trazas correspondientes a cada proceso e hilo de ejecución en un único fichero de traza que se pueda visualizar con Paraver.

Otras secciones del fichero de configuración permiten instrumentar las llamadas a CUDA, OpenCL, análisis de los elementos internos en los procesadores CELL y, en fase experimental, también existe la posibilidad de instrumentar la gestión sobre el uso de memoria dinámica y de entrada/salida de ficheros. Estas últimas no las hemos utilizado en nuestro perfilado de ejecución.

```
<?xml version='1.0'?>

<trace enabled="yes"
 home="/usr/local"
 initial-mode="detail"
 type="paraver"
 xml-parser-id="Id: xml-parse.c 2327 2013-11-22 11:47:07Z harald $"
>

  <!-- Configuration of some MPI dependant values -->
  <mpi enabled="yes">
    <!-- Gather counters in the MPI routines? -->
    <counters enabled="yes" />
  </mpi>

  <!-- Emit information of the callstack -->
  <callers enabled="yes">
    <!-- At MPI calls, select depth level -->
    <mpi enabled="yes">1-3</mpi>
    <!-- At sampling points, select depth level -->
    <sampling enabled="yes">1-5</sampling>
  </callers>

  <!-- Configuration of some OpenMP dependant values -->
  <openmp enabled="yes">
    <!-- Gather counters in the OpenMP routines? -->
    <counters enabled="yes" />
  </openmp>

  <!-- Configuration of User Functions -->
  <user-functions enabled="yes" exclude-automatic-functions="no">
    <!-- Gather counters on the UF routines? -->
    <counters enabled="yes" />
  </user-functions>

  <!-- Configure which software/hardware counters must be collected -->
  <counters enabled="yes">
    <cpu enabled="yes" starting-set-distribution="1">
      <set enabled="yes" domain="all" changeat-globalops="5">
        PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
        <sampling enabled="no" frequency="100000000">PAPI_TOT_CYC</sampling>
      </set>
      <set enabled="yes" domain="user" changeat-globalops="5">
       PAPI_TOT_INS,PAPI_FP_INS,PAPI_TOT_CYC
      </set>
    </cpu>
  </counters>

  <!-- Buffer configuration -->
  <buffer enabled="yes">
    <!-- How many events can we handle before any flush -->
    <size enabled="yes">500000</size>
    <circular enabled="no" />
  </buffer>

  <!-- Enable sampling capabilities using system clock. Type may refer to: default, real, prof and virtual. Period stands for the
sampling period (50ms here) plus a variability of 10ms, which means periods from  45 to 55ms.  -->
  <sampling enabled="yes" type="default" period="50m" variability="10m" />

  <!-- Do merge the intermediate tracefiles into the final tracefile?  -->
  <merge enabled="yes"
    synchronization="default"
    tree-fan-out="16"
    max-memory="512"
    joint-states="yes"
    keep-mpits="yes"
    sort-addresses="yes"
    overwrite="yes"
  />
</trace>
```

Extracto del fichero de configuración XML utilizado para la generación de trazas con Extrae

3.5. Ejecución del perfilado

Una vez compilada la aplicación AHF con soporte MPI y OpenMP, incluyendo los símbolos de depuración con el *flag* -g, procedimos a ejecutar Extrae para perfilar la versión original. A diferencia del fichero XML de configuración mostrado, el primer intento fue pasar, en el parámetro *list*, la ruta al fichero que contenía los nombres de las funciones de usuario que queríamos instrumentar. Esta ejecución provocaba una violación de segmento en la aplicación Extrae, concretamente en una sección que hace uso de la instrumentación automática de las funciones de usuario. Consultando a los desarrolladores de la aplicación, nos indicaron que el error podía deberse a una nueva funcionalidad que analizaba las llamadas a *fork* y que no estaba suficientemente probada. A pesar de que la aplicación AHF no hace ninguna llamada directa a esta función, obteníamos el siguiente error:

```
[kahan06:25749] *** Process received signal ***
[kahan06:25749] Signal: Segmentation fault (11)
[kahan06:25749] Signal code: Address not mapped (1)
[kahan06:25749] Failing at address: (nil)
[kahan06:25749] [ 0] /lib64/libpthread.so.0(+0xf500) [0x7fc012fe7500]
[kahan06:25749] [ 1] /opt/extrae/lib/lib_dyn_ompitracec-2.5.0.so(Clock_getLastReadTime+0x9)
[0x7fc00ff7bd29]
[kahan06:25749] [ 2] /opt/extrae/lib/lib_dyn_ompitracec-2.5.0.so(Extrae_Probe_fork_Entry+0x33)
[0x7fc00fcd0c93]
[kahan06:25749] [ 3]
/usr/lib64/dyninst/libdyninstAPI_RT.so(DYNINSTstaticHeap_16M_anyHeap_1+0x737) [0x7fc014513167]
[kahan06:25749] *** End of error message ***
```

Siguiendo las recomendaciones de los desarrolladores de Extrae, optamos por utilizar el API de Extrae incluyendo las llamadas a *Extrae_user_function()* en aquellas funciones de las que queríamos obtener información. Así mismo, pasamos a utilizar el mecanismo de la variable de entorno LD_PRELOAD (ver sección 2.2) para generar las trazas de ejecución de la aplicación. Este mecanismo consiste en asignar la librería con la que queremos generar las trazas a la variable de entorno LD_PRELOAD justo antes de lanzar la ejecución. En nuestro caso, las librerías que necesitamos utilizar son libmpitrace.so (para instrumentar las llamadas a MPI) o libompitrace.so (para instrumentar tanto las llamadas a MPI como a OpenMP), ambas proporcionadas por Extrae. En una ejecución no distribuida –sin MPI- puede ser trivial definir esta variable antes de invocar nuestra aplicación, pero en el caso paralelizado con MPI debemos hacer uso de un script que defina las variables de entorno necesarias por Extrae junto con el valor de LD_PRELOAD. Este script será invocado por *mpirun* seguido del nombre del ejecutable de nuestra aplicación y los argumentos que requiera.

Habiendo generado las trazas de ejecución en el clúster Kahan para el test “Halos going MAD” de 128^3 partículas, con 4 tareas MPI, y 4 threads OpenMP por tarea, visualizamos el siguiente comportamiento de la aplicación:

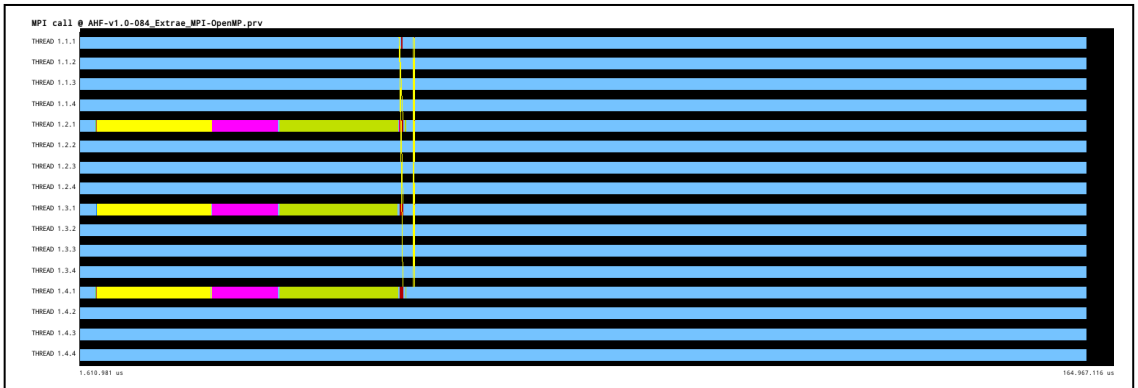

Figura 13. Diagrama de llamadas MPI con líneas de envío de datos. AHFv1

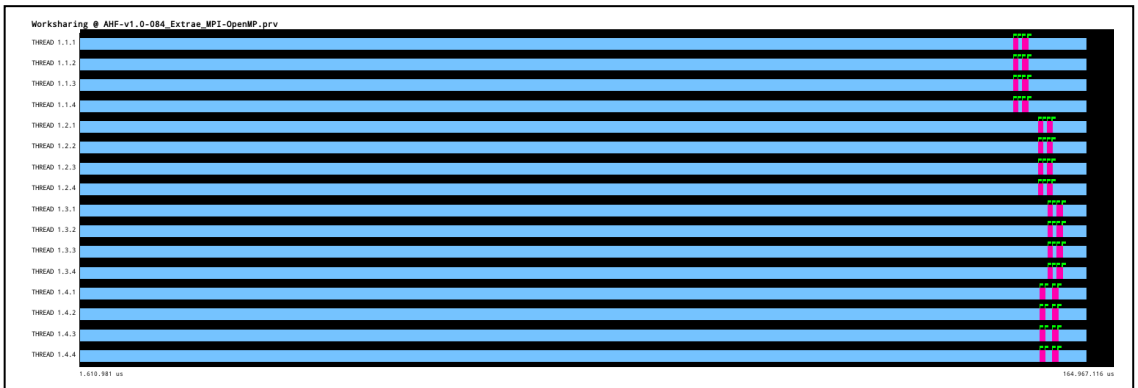

Figura 14. Diagrama de reparto de trabajo entre hilos de OpenMP, en fucsia las zonas que ejecutan regiones paralelas. AHFv1

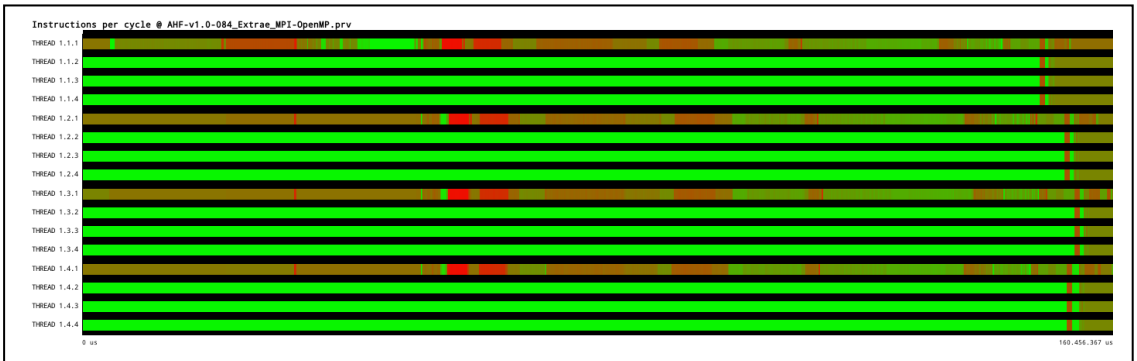

Figura 15. Diagrama de Instrucciones por ciclo en cada hilo (verde claro = 0,02; rojo intenso = 1,9). AHFv1

3.6. Identificación de región a paralelizar

Analizando los resultados del perfilado (ver Figuras 13, 14 y 15) podemos comprobar cómo aproximadamente el primer tercio del tiempo de ejecución se dedica a la lectura de los ficheros de datos, la descomposición de dominio y al intercambio de datos entre tareas MPI.

Una vez asignadas las partículas correspondientes a cada tarea MPI, comienza el refinamiento adaptativo, el descubrimiento de los patches como posibles halos, el cálculo de las propiedades físicas, el filtrado iterativo de partículas no ligadas gravitacionalmente y finalmente la determinación de los halos existentes, sus propiedades, la jerarquía de estructuras y la generación de los ficheros de salida.

Como puede observarse en el diagrama de reparto de trabajo entre hilos OpenMP, la paralelización intra-nodo se limita a dos regiones paralelas de muy limitada duración, aunque el balanceo de carga entre los hilos parece óptimo.

Teniendo en cuenta la complejidad y las limitaciones de la descomposición de dominio y el reparto de trabajo entre tareas MPI, abordar una modificación de esta parte supondría un trabajo de mayor envergadura que el que puede abordarse dentro de este TFM.

Por otra parte, los procesadores modernos incorporan cada vez mayor número de cores, siendo frecuente encontrar, en los supercomputadores actuales, nodos bi-procesador con un rango de núcleos por nodo entre 8 y 32. La capacidad de paralelización que ofrece la arquitectura actual parece no estar siendo aprovechada por la aplicación, por lo que nos centraremos en abordar la mejora de la paralelización OpenMP, una vez realizada la distribución de trabajo con MPI.

4. AHF versión 2

Modificaciones realizadas en el código para facilitar e implementar la paralelización intra-nodo con OpenMP

Habiendo analizado las estructuras de datos utilizadas en la versión 1 de AHF, hemos podido observar que siguen un esquema de lista enlazada, tanto para los QUAD como para las partículas. La dificultad de paralelizar este tipo de estructuras es bastante elevada, ya que no se conoce su tamaño con antelación (debiendo "saltar" entre punteros hasta encontrar un valor NULL) y su acceso puede requerir un uso intensivo de cerrojos (*locks*) para evitar condiciones de carrera entre distintos hilos OpenMP. Además, el patrón de acceso a memoria puede ser mayoritariamente aleatorio lo que no ayudará a reducir los fallos de cache. Esto podría evitarse reservando la memoria en bloques contiguos, pero parece algo complejo para tamaños tan grandes de memoria en un algoritmo de refinamiento adaptativo, en el que no se puede conocer con antelación las necesidades de memoria.

Por otra parte, el refinamiento adaptativo, que basa su decisión de continuar hacia niveles más finos en el número de partículas contenidas en una celda, implica una dependencia de datos ya que no se podrá refinar a un nivel dado hasta que no se complete el refinamiento en el nivel superior. También cabe resaltar que el cálculo en el refinamiento será poco intensivo, ya que consistirá en contar el número de partículas contenidas en una celda. En el caso en que se deba refinar, la carga computacional necesaria será básicamente la comparación de las posiciones de las partículas respecto a los límites de las celdas, lo que puede ser más costoso en términos de acceso a memoria que en operaciones en coma flotante, más aún cuando las estructuras que contienen las coordenadas de las partículas, al almacenar mucha más información (propiedades físicas) pueden tener un tamaño considerable. Por tanto, debe considerarse si la sobrecarga en la generación de hilos de ejecución OpenMP y la asignación de los mismos por parte del *runtime* a tareas de baja carga computacional generará o no alguna reducción en el tiempo total de ejecución.

Con el asesoramiento del desarrollador principal de AHF, nos hemos centrado en tratar de optimizar el refinamiento adaptativo partiendo de un array de estructuras que contenían las propiedades de las partículas. En concreto hemos utilizado las coordenadas de cada una de ellas, que es lo único necesario para realizar el refinamiento.

Considerando todo lo anterior hemos rediseñado las estructuras de datos para el refinamiento adaptativo.

4.1. Nuevas estructuras de datos

Son varias las estructuras que hemos utilizado en la nueva versión, siendo las más importantes el tipo abstracto de dato *cubekey*, utilizado como identificador único, codificado a nivel de bit,

para cualquier celda a cualquier nivel de refinamiento. También se ha hecho uso de implementaciones externas de tablas hash para almacenar las celdas en función de esta clave única, y de arrays dinámicos para almacenar las partículas correspondientes a cada celda y las celdas correspondientes a cada patch.

4.1.1. Cubekey

Valorando la necesidad de recorrer las distintas celdas que se irán generando a lo largo del refinamiento adaptativo hemos valorado principalmente dos opciones.

La primera y más intuitiva sería la generación de un octree, es decir, un árbol en que cada nodo contenga un puntero a cada uno de los ocho sub-nodos que puedan resultar de un refinamiento a un nivel más fino. Teniendo en cuenta la omnipresente utilización de sistemas operativos de 64 bits, más aún con requerimientos de memoria tan elevados por parte de la aplicación, debemos tener en cuenta que cada nodo en el octree debería contener al menos ocho punteros a sus nodos hijos, cada uno de los cuales ocupa 8 bytes (64 bits). Serían necesarios por tanto 64 bytes extra en cada una de las celdas del refinamiento, aún cuando no todos los punteros serán utilizados ya que el método adaptativo puede dar lugar a que una celda tenga menos de 8 sub-celdas. Además, cada una de las celdas deberá ser capaz de saber cuál es su posición y su tamaño o nivel de refinamiento, lo cual requeriría, al menos, 3 variables en coma flotante (un total de entre 12 –precisión simple- y 24 bytes –doble precisión-). El tamaño físico de la celda, como veremos más adelante, será fácilmente deducible sabiendo únicamente el nivel de refinamiento. Además, el recorrido de este octree tendría un acceso a memoria completamente irregular ya que la reserva de memoria de las celdas no podrá realizarse por bloques debido a su generación adaptativa. Este último problema podría solventarse parcialmente implementando un mecanismo de reserva de memoria por bloques y utilizando memoria de estos bloques para las sub-celdas que se generen de manera adaptativa, pero aún así no se garantizaría directamente la localidad espacial en la asignación de memoria a celdas físicamente próximas en la caja.

La segunda opción, finalmente implementada, ha sido la codificación de un identificador único (*cubekey*) capaz de mapear todas las celdas de la caja en una clave de 128 bits. La implementación original utilizaba 64 bits para codificar el identificador, pero como se explicará más adelante, esto limitaba el número de niveles de refinamiento a 21, lo cual no era suficiente para el tamaño de algunas de las simulaciones analizadas actualmente. A partir de esta clave única seremos capaces de determinar la posición y el tamaño de cada una de las celdas, así como la clave de la celda en el nivel superior de refinamiento (celda padre), las claves de todas las posibles sub-celdas (celdas hijas) y las claves de todas las celdas adyacentes (que serán candidatas para formar un patch). Al poder calcular los identificadores de las celdas padre e hijas, tendremos un diseño lógico similar a un octree, aunque sustituiremos el acceso inmediato mediante punteros por el cálculo de las cubekey correspondientes y su búsqueda en la tabla hash.

El diseño de esta clave está basado en la normalización de las posiciones de las partículas entre 0 y 1. La implementación está preparada, aunque no probada, para aceptar tamaños mayores,

pero teniendo siempre su origen en la posición (0, 0, 0). La ampliación del código para aplicar un *offset* en el caso de que el origen no fuese 0 no parece demasiado complicada.

Teniendo en cuenta que cada paso en el refinamiento consiste en la subdivisión de una celda origen en 8 sub-celdas, la identificación unívoca de cada una de estas 8 subdivisiones puede codificarse en tan solo 3 bits. Partiendo de la caja completa, cuyo origen de coordenadas sería la posición (0, 0, 0) y su tamaño de lado igual a 1, la división de esta caja daría lugar a 8 celdas en las que su posición vendrá determinada por su desplazamiento o la ausencia de este (bit a 1 o a 0), en cada una de las coordenadas. Así, para la celda comprendida entre los intervalos (0, 0.5) en el eje X, (0, 0.5) en el eje Y y (0.5, 1) en el eje Z, podría considerarse que hay ausencia de desplazamiento en los ejes X e Y, pero sí existe desplazamiento en el eje Z. La codificación del caso anterior en un *cubekey*, teniendo en cuenta que se codifican en orden XYZ, sería …01001 (todos los bits omitidos a la izquierda se consideran 0). Puede observarse que hay un 1 adicional delante de los 3 últimos bits. Este bit a 1, el primer bit a 1 desde la izquierda, es el flag que representa el nivel de refinamiento de la celda. El máximo número de niveles de refinamiento codificable en una clave de L bits será (L-1)/3. Debido a esta relación, el máximo número de niveles de una clave de 64 bits es 21. En la implementación con 128 bits, el número máximo de niveles de refinamiento es 42, estando este límite muy lejos de las necesidades de análisis de las simulaciones actuales. Cabe destacar, para visualizar el alcance de estos niveles de refinamiento, que teniendo en cuenta que cada subdivisión multiplica por 8 el número de celdas que componen la caja completa, en el nivel 21, si no utilizásemos un método adaptativo, generaríamos más de 9e+18 (9 trillones) celdas. En el nivel 42, el número llegaría a 8.5e+37.

Generalizando, el *cubekey* tendrá la siguiente estructura a nivel de bit:

$$\ldots 00 F X_n Y_n Z_n \; X_{n-1} Y_{n-1} Z_{n-1} \ldots \; X_3 Y_3 Z_3 \; X_2 Y_2 Z_2 \; X_1 Y_1 Z_1$$

A partir de esta información sabremos que estamos en el nivel n de refinamiento: contando el número de bits a cero comenzando por la izquierda, conoceremos la posición del *flag* F y así el nivel n. Existe una instrucción en los microprocesadores actuales (CLZ, *Count leading zeros*) que devuelven el número de ceros iniciales en una palabra (88) (su utilización podría acelerar una operación ejecutada una gran cantidad de veces). El tamaño de lado de la celda podrá calcularse fácilmente, siendo igual a $Box\_Size/2^n$. La posición del origen de la caja se obtendrá basándonos en los desplazamientos realizados en cada coordenada durante el refinamiento y multiplicándolo por el tamaño de la caja en ese nivel de refinamiento. El desplazamiento se calcularía, a nivel binario, de la siguiente manera:

$$Edge\_size = Box\_Size/2^n$$
$$X\_shift = X_1 \; X_2 \; X_3 \ldots X_{n-1} \; X_n$$
$$X\_pos = X\_shift \cdot Edge\_size$$
$$Y\_shift = Y_1 \; Y_2 \; Y_3 \ldots Y_{n-1} \; Y_n$$
$$Y\_pos = Y\_shift \cdot Edge\_size$$
$$Z\_shift = Z_1 \; Z_2 \; Z_3 \ldots Z_{n-1} \; Z_n$$
$$Z\_pos = Z\_shift \cdot Edge\_size$$

Con un ejemplo sencillo se aclarará el funcionamiento. Supongamos que tenemos una celda en el segundo nivel de refinamiento (n=2). Nos hemos desplazado en el eje X en las dos divisiones,

solamente en la primera división en el eje Y y solamente en la segunda división en el eje Z. La clave resultante sería la siguiente:

…01 101 110 (…0F $X_2Y_2Z_2$ $X_1Y_1Z_1$)

Edge_size = Box_Size/2^2 = 0.25 [asumiendo Box_Size=1]
X_shift = X_1 X_2 = $11_{base\ 2}$ = $3_{base\ 10}$
X_pos = 3 · 0.25 = 0.75
Y_shift = Y_1 Y_2 = $10_{base\ 2}$ = $2_{base\ 10}$
Y_pos = 2 · 0.25 = 0.5
Z_shift = Z_1 Z_2 = $01_{base\ 2}$ = $1_{base\ 10}$
Z_pos = 1 · 0.25 = 0.25

La celda se corresponde con la posición (0.75, 0.5, 0.25) y su tamaño de lado es 0.25.

El *cubekey* además permite calcular fácilmente las claves de otras celdas relacionadas, como son la celda padre, las celdas hijas producto del refinamiento y las posibles celdas adyacentes:

- Para calcular la clave de la celda padre, es decir, aquella con la que estaríamos conectados jerárquicamente en el nivel de refinamiento contiguo más grueso, bastará con poner a 0 los bits del último nivel de refinamiento y actualizar la posición del flag. En el ejemplo anterior:

  …01 101 110 (clave original) → …01 110 (clave padre)

- Siguiendo el funcionamiento explicado, la generación de las 8 posibles claves hijas correspondientes con el siguiente nivel de refinamiento (n=3) sería el siguiente:

  …01 101 110 (clave original) → …01 $X_3Y_3Z_3$ 101 110 (claves hijas)

En la descripción realizada del funcionamiento de la aplicación, puede intuirse la importancia que tendrá la detección de patches, es decir, partiendo de una celda dada, conocer cuáles son las celdas adyacentes que formarán esos bloques aislados de celdas. Habiendo revisado el cálculo de los desplazamientos en cada una de las coordenadas codificados en la *cubekey*, se puede observar cómo el reordenamiento de los bits de desplazamiento hace que, por ejemplo, un 1 en la posición X_1 suponga un desplazamiento en el nivel n=1, dos desplazamientos en el nivel n=2, cuatro en el nivel n=3, etcétera.

- El cálculo de la clave de una celda adyacente se puede ver como aquella cuyo desplazamiento en una, y sólo una, de sus coordenadas se incrementa o decrementa en una unidad. Teniendo en cuenta las 3 coordenadas del espacio, estas claves representarían las celdas que comparten una cara de la celda. Podremos obtener así las claves de las celdas unidas por una de las 6 caras. Si consideramos las combinaciones de desplazamientos en dos de las tres coordenadas, estaremos obteniendo aquellas celdas unidas por las 12 aristas del cubo. El siguiente paso lógico

sería considerar aquellas celdas unidas por las esquinas o vértices, que implicará un desplazamiento en las tres coordenadas, añadiendo otras 8 posibles celdas adyacentes. Una vez que, partiendo del identificador de una celda, hayamos extraído sus desplazamientos en cada una de las coordenadas, podremos construir los desplazamientos correspondientes a las celdas adyacentes. Aplicando el procedimiento inverso, podremos combinar los desplazamientos calculados de las celdas adyacentes para obtener sus identificadores.

En este momento corresponde explicar el concepto clave para la implementación de las condiciones de periodicidad en la caja de la simulación, en lo que respecta a las celdas (otras consideraciones extraordinarias son requeridas al calcular, por ejemplo, distancias entre partículas o centros de masa, pero no son objeto de este trabajo). Continuando con el ejemplo, hemos visto cómo el desplazamiento en el eje X era el máximo posible en el nivel 2. Una vez extraído el *X_shift* ($3_{base\ 10}$, $11_{base\ 2}$) deberíamos proceder a incrementarlo y decrementarlo, pero no es posible incrementar el desplazamiento, ya que obtendríamos un valor de 4, que multiplicado por el tamaño del lado daría una celda cuyas coordenadas en el eje X se encontrarían entre 1 y 1.25, es decir, fuera de la caja. Aplicando la operación módulo con el máximo desplazamiento posible por eje (2^2 para n=2), obtendríamos un desplazamiento incrementado igual a 0 y uno decrementado igual a 2. Esto implicaría que, partiendo de la celda cuyos límites en el eje X son (0.75, 1), la celda adyacente incrementada se encontraría en (0, 0.25), debido a la condición de periodicidad, y la decrementada en (0.5, 0.75). Extrapolando este comportamiento a las 3 coordenadas del espacio conseguimos calcular las claves de las celdas adyacentes aplicando las condiciones de periodicidad requeridas.

Una vez diseñada esta clave única para cada celda del refinamiento adaptativo, es fundamental la implementación de la rutina que, a partir de las coordenadas espaciales y el nivel de refinamiento, genere el *cubekey* a nivel de bit. El siguiente pseudocódigo muestra el funcionamiento de esta rutina clave, llamada coor2ck.

Como se puede apreciar en el pseudocódigo, se inicializa el tamaño de lado al tamaño de la caja (edge_length=Box_size), y las coordenadas de la celda que vamos calculando (cur_x=cur_y=cur_z=0). Después, para cada nivel de refinamiento, comprobamos si debemos o no desplazarnos comparando las coordenadas de la celda con las coordenadas recibidas. En el caso en que debamos desplazarnos, ponemos a 1 el bit correspondiente a la coordenada en su nivel de refinamiento y actualizamos la coordenada de la celda.

Pseudocódigo de coor2ck

```
coor2ck (ckey, xcoor, ycoor, zcoor, depth){

        edge_length=Box_Size;
        cur_x=cur_y=cur_z=0;

        for (i=0; i<depth; i++){
                half=edge_length/2;
                //Clear previous depth-FLAG
                clr_bit_ck(ckey,3*(i));
                //Set Depth-FLAG
                set_bit_ck(ckey,3*(i+1));

                if (xcoor >= cur_x + half){
                        cur_x+=half;
                        set_bit_ck(ckey,3*i+2);
                }

                if (ycoor > cur_y + half){
                        cur_y+=half;
                        set_bit_ck(ckey,3*i+1);
                }

                if (zcoor > cur_z + half){
                        cur_z+=half;
                        set_bit_ck(ckey,3*i);
                }
                //Update edge_length for the next iteration
                edge_length=half;

        }
}
```

Durante las pruebas de la implementación observamos que los resultados físicos de considerar como adyacentes aquellas celdas que únicamente compartían un vértice o esquina de la celda provocaban efectos distorsionadores, considerando como un único patch dos patches independientes con posiciones cercanas. La implementación definitiva, por tanto, únicamente considera las 18 celdas adyacentes que comparten una cara (6) o una arista (12) con la celda original, y se descartan las 8 celdas adyacentes que únicamente compartan un vértice (ver Figura 16).

Llegados a este punto, contamos con una manera de identificar unívocamente cada una de las celdas, en cualquier nivel de refinamiento, utilizando 128 bits (16 bytes). Además hemos revisado la posibilidad de conocer el identificador correspondiente a sus posibles celdas hijas, a su padre y a sus adyacentes. Es cierto que no contamos con punteros que nos proporcionen acceso directo a estas celdas relacionadas, pero estamos utilizando 16 bytes para la clave, cuando solamente los 8 punteros a las posibles celdas hijas ocuparían 64 bytes (8 punteros, 8 bytes por puntero).

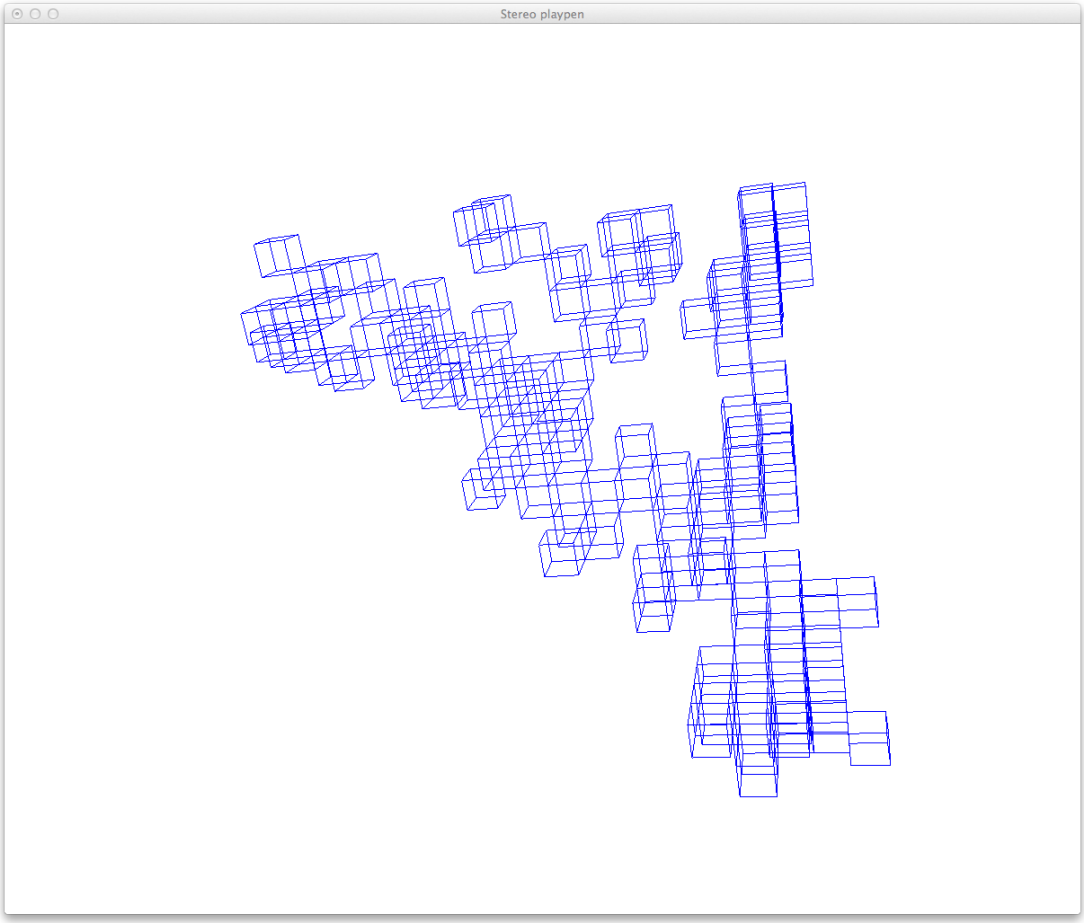

Figura 16. Visualización de la distribución de celdas adyacentes formando un patch. A. Knebe

El reto ahora consiste en encontrar la forma de almacenar y buscar estas celdas, basándonos en su identificador, de una manera eficiente respecto al uso de memoria y rápida a la vez.

4.1.2. Tablas hash y arrays dinámicos

A la vez que se define el tipo de dato *cubekey* con todas las características ya explicadas, se plantea qué estructura de datos puede ser la más adecuada para almacenar y acceder estas celdas a partir de su clave. La posibilidad de un árbol binario, y en concreto su extensión a un octree, se descarta debido al sobrecoste de uso de memoria y al patrón de acceso aleatorio que implicaría.

La opción inicial que se valoró fue utilizar un árbol balanceado (89) y posteriormente un árbol B (90), que permite reducir el número de nodos al almacenar varios valores en cada uno. El principal inconveniente que planteaban las estructuras anteriores era su característica de mantener los nodos ordenados, lo que permite un tiempo de acceso logarítmico, a costa de realizar frecuentes rebalanceos de los nodos. En nuestro caso concreto, no teníamos ningún interés en que las claves se encontrasen ordenadas, ya que el orden en las *cubekey* no implica necesariamente la proximidad espacial de las celdas correspondientes. Por tanto, un recorrido

ordenado de los valores almacenados no aportaría ninguna ventaja para los algoritmos que se iban a utilizar.

La última opción considerada y finalmente adoptada fue la utilización de tablas hash (91). No es objeto de este trabajo analizar y describir el funcionamiento de estas estructuras de datos, pero asumiremos un conocimiento medio sobre las mismas por parte del lector. Serán fundamentales, para el tiempo de ejecución de los algoritmos que hagan uso de estas tablas, conceptos básicos como la función hash utilizada (92), el número de posiciones disponibles (*buckets*) correspondiente al rango de valores que puede generar la función, la resolución de colisiones y la forma de almacenar varias claves para una misma posición (valores que colisionan).

Habiendo descartado la opción de desarrollar nuestra propia implementación de tablas hash –por limitaciones de tiempo y por la urgencia de obtener un prototipo funcional lo antes posible- y decidiendo ceñirnos al lenguaje C utilizado en la aplicación AHF –lo que nos impidió utilizar la clase de C++ *unordered_map* (93), añadida al STL (94) en la última revisión de 2011 y que implementa internamente una tabla hash (95)- optamos por utilizar la implementación realizada por Troy D. Hanson en su proyecto uthash (96). Este proyecto permite, de manera muy sencilla, almacenar cualquier estructura definida en C en una tabla hash. Para ello se debe incluir el fichero uthash.h, que define las estructuras de datos internas necesarias así como las rutinas para inserción, acceso y demás funcionalidad proporcionada. También se deberá incluir, dentro de la estructura que queramos almacenar en la tabla, una variable de tipo *UT_hash_handle* (definida en uthash.h). Sin querer entrar en los detalles de implementación de uthash ni de utilización de sus rutinas, cabe mencionar que todas las funciones que se proveen se implementan como macros en el fichero *header* proporcionado, y que la sobrecarga de memoria que representa la inclusión de la variable *UT_hash_handle* es de 56 bytes por cada estructura según la documentación del proyecto (97) (la gestión de los *buckets*, contadores y punteros internos también supondrá un pequeño sobrecoste de memoria). Efectivamente, esta sobrecarga, cuando el número de estructuras (celdas resultantes del refinamiento adaptativo) va a ser muy elevado, puede arruinar nuestros esfuerzos por reducir el consumo de memoria en la aplicación. Pero también es cierto que no solamente buscamos la reducción de consumo de memoria, sino el diseño de un algoritmo que permita la paralelización intra-nodo mediante OpenMP. Si los resultados de la paralelización fuesen favorables, podríamos, en un trabajo futuro, buscar otra implementación de tablas hash o realizar la nuestra propia.

Por otra parte, debe considerarse que cada celda, a lo largo del refinamiento adaptativo, debe almacenar internamente las partículas que le correspondan, o más concretamente, un puntero a las mismas. Almacenar el identificador de la partícula, equivalente a su posición en el array de partículas original, requeriría el mismo tamaño en memoria (64 bits) pero acceder a ella implicaría una operación aritmética para calcular su posición a partir del inicio del array, por lo que optamos por almacenar el puntero directamente. Dado que el número de partículas puede llegar a ser muy elevado y a la vez muy diferente, podríamos implementar una lista enlazada para almacenarlas, pero no parece la opción más eficiente si la lista alcanza un gran tamaño. Otra opción habría sido utilizar un array y ampliar su tamaño según fuese necesario, lo cual implicaría numerosas llamadas a la función *realloc* (98), con sus correspondientes llamadas al sistema y posible necesidad de mover grandes bloques de memoria. Sacrificando la búsqueda

de una opción más eficiente, optamos por utilizar los *utarray* (99), que forman también parte del proyecto uthash. Internamente, su funcionamiento es equivalente a un array que se amplía mediante llamadas a *realloc*, pero provee las rutinas de inserción y extracción que se hacen cargo de la gestión de memoria. Además, una de las rutinas (*utarray_reserve* (100)), permite la reserva de memoria para *n* elementos más, de manera que podría reducirse el número de veces que se redimensiona el array reservando memoria por bloques en vez de para cada elemento. Esta mejora aún no se ha implementado en la versión 2 de AHF.

Todo lo explicado anteriormente respecto al almacenamiento de punteros a partículas dentro de cada celda es equivalente para el caso de los patch, en los cuales deben almacenarse punteros a las celdas que los forman.

La disponibilidad por parte de Troy D. Hanson respondiendo a las preguntas sobre su código, a través de la plataforma GitHub, ha sido fundamental: surgió un error de compilación al utilizar los *utarray* que identificamos y resolvimos (101). También tuvimos que comprobar el máximo número de elementos que es capaz de almacenar su implementación de tablas hash, dado que nuestras necesidades pueden alcanzar fácilmente los miles de millones de elementos por tabla, y el desarrollador nos comentó que él nunca había requerido almacenar miles de millones de elementos, aunque sí había probado con varios millones. Revisando el código fuente, nos percatamos de que el máximo número de elementos que puede almacenar una tabla podría estar limitado por el identificador de 32 bits utilizado internamente en cada *bucket*, así como por el número de *buckets* que se generen. La implementación realizada por Hanson incluye un rebalanceo de los elementos aumentando el número de *buckets* cuando el número de "colisiones" es muy alto, pero desconocemos los detalles de su funcionamiento interno. Por tanto, optamos por realizar una prueba tratando de insertar 2^{33} elementos para comprobar así si el límite estaba en la utilización de un entero sin signo de 32 bits. Obtuvimos un error de violación de segmento al insertar algo más de dos mil millones de elementos, próximo al valor de 2^{31} (102). El incidente sigue abierto y pendiente de encontrar una solución, pero nosotros hemos podido hacer uso de las tablas para nuestra implementación sin encontrar problemas debido a esta limitación.

4.2. Algoritmo de refinamiento adaptativo para AHFv2

Tras describir las estructuras de datos utilizadas en la implementación del refinamiento adaptativo de la versión 2 de AHF, pasaremos a describir cómo se ha diseñado el algoritmo que hace uso de las mismas y permite su paralelización intra-nodo. Debemos recordar que el punto de partida para nuestra implementación es un array de partículas del que conocemos su dirección de memoria inicial y su tamaño, por lo que podremos recorrerlo. Y al conocer la estructura de datos de las partículas, podremos acceder a sus posiciones dentro de la caja. El otro parámetro que necesitamos para realizar el refinamiento es el número de partículas umbral que nos indicará si una celda debe ser refinada o no.

El primer paso será recorrer cada una de las partículas del array y calcular el *cubekey* correspondiente a esa posición, buscar en la tabla hash la celda correspondiente a la clave

calculada, y en el caso de que no exista crearla e insertarla en la tabla. Después, procederemos a registrar la partícula en el array dinámico de la celda que le corresponda.

En la caja resultante de una simulación, la división inicial en 8 celdas de la caja completa puede aportar información de poca relevancia a la hora de descubrir estructuras cosmológicas. Con este propósito, y basado en parámetros físicos de la simulación, el desarrollador principal de la aplicación nos proporcionó una rutina (*ptree_min_level*) que nos devuelve el nivel más grueso en el que debemos comenzar el refinamiento adaptativo. De esta manera ahorraremos tiempo al evitar realizar subdivisiones innecesarias en niveles superiores, y también ahorraremos memoria al evitar generar celdas sin relevancia.

Un aspecto fundamental que nos va a permitir la paralelización del código es la utilización de una tabla hash independiente en cada nivel de refinamiento. De este modo, una vez que completemos el refinamiento en un nivel, aquellas celdas que deban ser refinadas, basándonos en el número de partículas umbral, generarán únicamente las celdas no vacías del nivel inmediatamente más fino. Se traspasarán las partículas del array dinámico de la celda refinada a los arrays dinámicos de las celdas hijas resultantes para evitar replicar punteros a partículas en todas las celdas por las que vayan "pasando" estas partículas. Y como se indicaba anteriormente, las celdas hijas generadas se guardarán en la tabla hash correspondiente a su nivel de refinamiento, diferente del nivel de la celda padre. Para implementar este conjunto de tablas hash, se creará un array de punteros a tablas hash, cuyo tamaño será el máximo nivel de refinamiento que se pueda alcanzar (42 para *cubekey* de 128 bits). Aquellas tablas hash que no se utilicen supondrán un desperdicio de 8 bytes de memoria (su puntero a NULL), pero a cambio tendremos una separación de datos por niveles de refinamiento, lo que nos permitirá paralelizar cualquier operación que requiera acceso a las celdas de un único nivel. Más adelante veremos cómo este es precisamente el caso de la búsqueda de los patches, que además supone el mayor coste computacional.

La implementación de esta versión 2 ha requerido la creación de una librería llamada libtree, en la cual se han encapsulado todas las definiciones de estructuras y funciones que permiten trabajar con las *cubekeys*, los *subcubes* (celdas) y los patches. La implementación de las funciones se ha realizado intentando abstraerse de la utilización interna de uthash y utarray, para facilitar la modificación y optimización en el futuro. El punto de entrada a libtree desde el código principal de la aplicación (desde la función *main* concretamente) es la función generate_tree. No menos importante ha sido el desarrollo de la librería libahf2 (adaptación de libahf) por parte de Alexander Knebe, la cual realiza la detección de halos y cálculo de propiedades físicas de los mismos, basándose en el nuevo formato de árbol de patches que libtree genera. El código de estas librerías es código abierto y puede descargarse de la web de AHF (36).

Aunque probablemente a estas alturas el lector pueda imaginarse la estructura del algoritmo que realiza el refinamiento adaptativo y la búsqueda de patches, vamos a intentar describirlo sin entrar en detalles de implementación.

El pseudocódigo que describiría el funcionamiento desde la lectura del array de partículas hasta la generación del árbol de patches sería el siguiente.

Pseudocódigo de generate_tree

```
generate_tree(num_particles, first_particle, threshold, min_part_patch){
  //Una tabla hash por cada nivel de refinamiento
  subcube_table[depth] (depth 1..42)
  //Se obtiene el nivel en el que se debe comenzar a refinar
  initial_depth=ptree_min_level()

  //División inicial: se crean las celdas no vacías correspondientes al nivel initial_depth
  initial_box_division(npart, fst_part, initial_depth, subcube_table);

  /* Recorremos todas las celdas de la tabla subcube_table[initial_depth], refinando aquellas
   * con más de NtreeMin partículas. Después repetimos la operación para las tablas de los
   * siguientes niveles hasta que lleguemos al nivel máximo o hasta que encontremos una tabla
   * vacía.*/
  refine_box_division(initial_depth, subcube_table, threshold);

  //En este punto el refinamiento adaptativo ha terminado. Procedemos a buscar los patches.
  patches_generation(&patches, subcube_table, NminPerHalo);

  //Conectamos la jerarquía de patches entre los distintos niveles
  patch_connect_tree(patches->tree, subcube_table);

  //Se calculan parámetros físicos del árbol/bosque de patches
  set_patch_pos(patches->tree, patches->n_patches);
  set_patch_trunk(patches->tree, patches->n_patches);
  set_patch_radii(patches->tree, patches->n_patches);

  /* Liberamos la memoria de las tablas, subcubos y listas de partículas.
   * Toda la información necesaria está en los patches.*/

  return patches;
}
```

- initial_box_division: Se recorre el array de estructuras de partículas, calculando para cada una su celda correspondiente y guardando en ella un puntero a la partícula. Las celdas generadas se guardan en la tabla hash correspondiente al nivel inicial de refinamiento (subcube_table[initial_depth]).
- refine_box_division: Una vez generadas las celdas en el nivel inicial, las recorreremos todas refinando aquellas con un número de partículas superior al umbral (*threshold*). El refinado implicará calcular la clave *cubekey* de las partículas en el nivel inmediatamente más fino, crear la celda si no existe y registrar en ella la partícula, y almacenar la celda en la tabla hash del nivel correspondiente. Una vez que se termine de refinar un nivel, se recorrerá la tabla hash del siguiente: si se refinó alguna celda en el nivel más grueso, habrá celdas cuyo número de partículas tendremos que revisar. Repetiremos este procedimiento hasta que encontremos una tabla hash vacía, lo que significará que el nivel anterior es el más fino que se ha alcanzado. Una vez llegados a este punto habremos finalizado el refinamiento adaptativo.
- patches_generation: En este momento procedemos a buscar, en cada nivel de refinamiento, grupos de celdas adyacentes que formen patches. Recorreremos la tabla hash de celdas de un nivel y para cada celda, si no tiene ya un patch asignado, procederemos a buscar todas las posibles celdas adyacentes en la tabla que formarán el nuevo patch. Para aquellas celdas contiguas que encontremos en la tabla, a su vez, deberemos comprobar sus adyacentes. Continuaremos con este procedimiento, intuitivamente recursivo, hasta que no encontremos nuevas celdas adyacentes sin patch asignado. Estos patches deberán tener al menos *NminPerHalo* partículas, parámetro proporcionado por el usuario, para que se consideren como un posible halo.

- patch_connect_tree: Una vez detectados los patches en cada nivel de refinamiento, procedemos a conectar cada patch con su padre, es decir, el patch que lo contiene en el nivel más grueso. Por el diseño del algoritmo y las estructuras de datos utilizadas, bastará tomar una celda de un patch, calcular el *cubekey* de la celda padre de la manera explicada anteriormente y comprobar a qué patch pertenece esa celda padre. A su vez, nos registraremos en el patch padre como hijos. Así, una vez terminado este procesamiento, cada patch tendrá conocimiento de cuáles son sus patches hijos y cuál es su padre, con lo que tendremos el árbol de patches que debemos proporcionar

La función más problemática de desarrollar fue *patches_generation*, debido a que la implementación intuitiva parece requerir un comportamiento recursivo: al partir de una celda que aún no pertenece a ningún patch, parece lógico buscar en la tabla hash los *cubekey* de las celdas adyacentes. En el caso de encontrar alguna de estas celdas en la tabla, volveríamos a hacer la misma búsqueda partiendo de las celdas contiguas descubiertas, y así recursivamente hasta que no encontremos ninguna celda adyacente no visitada anteriormente. Además de la sobrecarga en el uso de la pila de llamadas que tiene cualquier implementación recursiva, en nuestro caso concreto, el gran número de llamadas recursivas que se realizaba alcanzaba tal profundidad que desbordaba la memoria de la pila de llamadas y provocaba una violación de segmento. Analizando el fichero core generado, nos percatamos de que se alcanzaban varias decenas de miles de llamadas recursivas. Aún ampliando el tamaño de la pila de llamadas a varios gigabytes con el uso de la variable de entorno OMP_STACKSIZE, seguíamos obteniendo el mismo error. Esto es lógico, ya que, el número de posibles cubos que se pueden generar en un nivel de refinamiento no demasiado profundo es enorme: solo en el decimoquinto nivel el número de celdas máximo es de más de 35 billones (8^{15}). Si, siguiendo con el ejemplo, suponemos que tenemos un patch que ocupa una décima parte de la caja, estará formado por aproximadamente 3.5 billones de celdas. En el mejor de los casos, si comenzásemos la búsqueda recursiva desde la celda que ocupa la posición central del patch y éste tuviese una forma esférica homogénea como caso más favorable, tendremos que recorrer la mitad del número de celdas en cada dimensión. Siendo la raíz cúbica de 3.5 billones de celdas aproximadamente 15.000, la profundidad que alcanzaría la pila de llamadas recursivas sería de 7.500. Todo lo anterior considerando como el mejor de los casos de un patch homogéneo en el nivel 15.

Por este motivo tuvimos que modificar la implementación para romper la recursividad y desarrollar un método iterativo que buscase celdas adyacentes, de manera similar al recorrido de un árbol en anchura, frente al método recursivo similar a un recorrido en profundidad. Para ello implementamos un array dinámico de *cubekeys*, llamado *visitable_cubekeys* en el que almacenamos las claves de las celdas adyacentes que debemos buscar en la tabla. Cada vez que se extrae una clave de este array dinámico se comprueba si la celda correspondiente existe y aún no ha sido visitada y, en ese caso, se añaden en otro array dinámico auxiliar, también de tipo *visitable_cubekeys*, las claves de las celdas adyacentes que deberemos visitar en la siguiente iteración. Una vez hayamos buscado todas las claves de la estructura principal, tendremos en la estructura auxiliar las claves de las celdas contiguas que debemos buscar en la tabla. Intercambiando la estructura principal por la auxiliar podremos proceder iterativamente.

Cuando hayamos terminado de visitar todas las celdas que formen un patch, las dos estructuras *visitable_cubekeys* estarán vacías.

Tal y como hemos diseñado el algoritmo, parece claro que existen ciertas dependencias de datos entre las distintas regiones. Por ejemplo, no podremos realizar el refinamiento adaptativo hasta que no hayamos terminado con el refinamiento inicial de la caja. Tampoco se podrá comenzar la búsqueda de patches hasta que no se hayan generado las celdas que pueden formarlos. Obviamente, tampoco podremos conectar los patches en forma de árbol hasta que no los hayamos encontrado todos. Algunas de estas restricciones, como la necesidad de generar las celdas para buscar los grupos aislados que forman patches, son intrínsecas al problema. Otras, en cambio, han sido impuestas por nuestra implementación y podrían ser replanteadas para facilitar la paralelización en un futuro. Por ejemplo, una vez generadas las celdas en un nivel dado de refinamiento, dentro de la función *refine_box_division*, podría comenzar la búsqueda de patches en ese nivel a la vez que continúa el posible refinamiento en niveles más finos.

4.3. Medidas de tiempos en implementación secuencial

Una vez que hemos implementado la versión secuencial del algoritmo de generación del árbol de patches anteriormente descrito, podríamos comenzar a intentar paralelizar cada una de las funciones internamente, pero es probable que las paralelizaciones más triviales no sean las que reduzcan los mayores costes computacionales. Por lo tanto, optamos por medir tiempos de ejecución de cada una de las funciones mediante llamadas a la función OMP_get_wclock.

Para el caso de test correspondiente a CLUES, cuya ejecución completa toma entre 50 y 60 minutos, se puede ver en los fragmentos del log de ejecución cómo el tiempo dominante es el empleado en la sección *set_patch_pos/trunk/radii* con 939 segundos, correspondiente al cálculo de parámetros físicos y ejecutado posteriormente al refinamiento adaptativo y a la generación del árbol de patches. Esta sección es por tanto a nuestro trabajo. La siguiente región que más cómputo requiere es *patches_generation*, con 719 segundos. Después estarían, en orden descendiente de tiempo de ejecución, *refine_box_division* (285 segundos), *initial_box_division* (22 segundos) y *patches_connect_tree* (0.29 segundos). Viendo estas cifras, optamos por intentar paralelizar la generación de los patches, ya que es la región que más tiempo consume y además su paralelización, como veremos seguidamente, es bastante sencilla ya que no requiere ningún mecanismo de sincronización y protección de acceso adicional como el cerrojos de lectura/escritura que podrían ser necesarios si varios hilos de ejecución accediesen a tablas hash compartidas al paralelizar, por ejemplo, el refinamiento adaptativo de la caja o la división inicial.

Es también muy orientativa la información contenida en el log sobre el número de celdas contenidas en cada una de las tablas hash (HASH_COUNT) correspondiente a cada nivel. Puede observarse cómo, en los niveles 12 a 15, el número de celdas se encuentra entre 10 y 28 millones. Por otra parte, la mayoría de los niveles de refinamiento (del 9 al 11 y del 16 al 19)

contienen entre 1 y 7 millones de celdas. A partir del nivel 20 no se alcanza el millón de celdas por nivel, llegando en el último nivel de refinamiento a tener tan solo 12 celdas.

La información sobre la construcción de los patches en los distintos niveles muestra que una gran cantidad de patches, una vez construidos, son descartados (*rejected*) por no cumplir el mínimo número de partículas necesario para considerar un patch como posible halo. Por ejemplo, en el nivel 13, que es el que cuenta con un mayor número de celdas, se generan 31029 patches y se descartan 365017. También puede observarse cómo el tiempo empleado en la generación de patches en los distintos niveles, debido al diferente número de celdas, es tremendamente irregular. De los 17 diferentes niveles en los que existen celdas (del 9 al 25), la búsqueda de patches en los dos últimos, que son los menos poblados, no requiere siquiera una décima de segundo. En cambio, en los niveles más poblados (del 12 al 15), los tiempos requeridos oscilan entre los 80 y los 200 segundos.

Teniendo en cuenta que la búsqueda de patches por nivel es una tarea completamente independiente y sin ninguna dependencia de datos, hemos optado por paralelizar esta región.

Fragmentos de logs generados por AHFv2 (versión secuencial)

```
 User Input:
=============
ic_filename       = /data2/AHFv2/tests/AHF/00DATA/CLUES-WM3/CLUES-B64_WMAP3_186592_GAS_SFR-z0.000 (Gadget
binary, 60)
[…]

### Executing with NEWAMR ###
Max particles per subcube: NtreeMin=3
ptree_min_level: determining minimum level for patch-tree:
[…]
DEPTH returned by ptree_min_level = 9
Initial box division...
initial_box_division() at ptree_min_level depth (9) generated 1626277 subcubes
=timing= initial_box_division() took 22.727316 seconds
Refine recursively...
REFINEMENT IS OVER (initial_depth=9, final_depth=25).
=timing= refine_box_division() took 285.354807 seconds
Depth level= 9 (       512 div per dim), HASH_COUNT=   1626277, stored_particles=   1654566
Depth level=10 (      1024 div per dim), HASH_COUNT=   1662901, stored_particles=   1206496
Depth level=11 (      2048 div per dim), HASH_COUNT=   5868816, stored_particles=   4890957
Depth level=12 (      4096 div per dim), HASH_COUNT=  17764991, stored_particles=  20112349
Depth level=13 (      8192 div per dim), HASH_COUNT=  28229226, stored_particles=  35853330
Depth level=14 (     16384 div per dim), HASH_COUNT=  18363211, stored_particles=  22806989
Depth level=15 (     32768 div per dim), HASH_COUNT=  10748201, stored_particles=  13648057
Depth level=16 (     65536 div per dim), HASH_COUNT=   7026624, stored_particles=   9074441
Depth level=17 (    131072 div per dim), HASH_COUNT=   3832228, stored_particles=   4928507
Depth level=18 (    262144 div per dim), HASH_COUNT=   2139373, stored_particles=   2717745
Depth level=19 (    524288 div per dim), HASH_COUNT=   1416720, stored_particles=   1814323
Depth level=20 (   1048576 div per dim), HASH_COUNT=    849655, stored_particles=   1090631
Depth level=21 (   2097152 div per dim), HASH_COUNT=    503009, stored_particles=    644813
Depth level=22 (   4194304 div per dim), HASH_COUNT=    301929, stored_particles=    405494
Depth level=23 (   8388608 div per dim), HASH_COUNT=     96259, stored_particles=    126939
Depth level=24 (  16777216 div per dim), HASH_COUNT=      4189, stored_particles=      5146
Depth level=25 (  33554432 div per dim), HASH_COUNT=        12, stored_particles=        25

Building the patches...
Building the patches at level 9
We built 5329 patches (614111 rejected) at level 9 in 14.345183 seconds
Building the patches at level 10
We built 3357 patches (19183 rejected) at level 10 in 10.732023 seconds
Building the patches at level 11
We built 3462 patches (25626 rejected) at level 11 in 37.203046 seconds
Building the patches at level 12
We built 7817 patches (86602 rejected) at level 12 in 124.561111 seconds
Building the patches at level 13
We built 31029 patches (365017 rejected) at level 13 in 200.969867 seconds
Building the patches at level 14
We built 51587 patches (587216 rejected) at level 14 in 137.242004 seconds
Building the patches at level 15
We built 41823 patches (401283 rejected) at level 15 in 80.067245 seconds
Building the patches at level 16
We built 32168 patches (272269 rejected) at level 16 in 45.642629 seconds
Building the patches at level 17
We built 19955 patches (160992 rejected) at level 17 in 32.196228 seconds
[…]
Building the patches at level 24
We built 6 patches (1071 rejected) at level 24 in 0.019644 seconds
Building the patches at level 25
We built 0 patches (6 rejected) at level 25 in 0.000051 seconds
All (and ONLY) the calls to patches_generation_per_level() took 719.732004 seconds

=timing= patches_generation() took 719.732304 seconds
Connecting patches between parents and daughters
Patches->tree connection has finished.
=timing= patches_connect_tree() took 0.290382 seconds
=timing= set_patch_pos/trunk/radii took 939.746245 seconds
=timing= Freeing memory of UThash tables of subcubes and internal UTarrays of particles took 65.564941
seconds
```

4.4. Paralelización de la búsqueda de patches

La versión secuencial de AHFv2, a la hora de generar los patches, recorre los distintos niveles mediante un bucle *for*, y en cada iteración/nivel se invoca a la misma función (*patches_generation_per_level*) encargada de detectarlos y almacenarlos.

Ya hemos visto que no existe dependencia de datos de lectura para la construcción de los patches en distintos niveles de refinamiento, pero deberemos asegurarnos también de que no existan posibles conflictos al almacenar los patches generados. Para ello hemos creado una estructura de datos llamada *ahf2_patches* que nos va a permitir acceder de manera independiente, basándonos en la posición de sus arrays internos, a la información propia de cada nivel. Esta estructura contiene un array de punteros llamado *tree*, de la misma longitud que posibles niveles puede haber (en el caso de una cubekey de 128 bits, el máximo será 42 niveles). Los punteros son de tipo *patch_t*, que es la estructura de datos en la que se almacena la información de las celdas que componen el patch y una gran cantidad de parámetros físicos. Además, *ahf2_patches* tiene otros dos arrays, también de longitud igual al número de posibles niveles, que contabilizarán, en cada nivel/posición, el número de patches que se han generado y que se han descartado. En cada posición del array de punteros *tree* se irá reservando memoria dinámicamente para los nuevos patches que se vayan generando y almacenando. Este mecanismo no óptimo puede implicar un gran número de llamadas a la función *realloc* y el diseño en forma de "estructura de arrays" podría provocar múltiples fallos de cache al producirse accesos, por distintos hilos de ejecución, a regiones de memoria contiguas, pero nos provee de un acceso de escritura independiente para cada hilo y por tanto evitamos la utilización de cerrojos que protejan posibles conflictos.

La región que hemos paralelizado con OpenMP está formada por el bucle *for* mencionado anteriormente, que se encarga de llamar a la rutina que genera los patches en cada nivel. Es importante resaltar que el número de iteraciones de este bucle es muy limitado, y la carga computacional de cada una es muy diferente, directamente relacionada con el número de celdas por nivel. Por este motivo hemos utilizado la sentencia de OpenMP *#prama omp parallel for* configurando el *scheduler* como dinámico y un tamaño de *chunk* igual a 1, mediante la opción *schedule(dynamic,1)*. De esta forma, asignamos una única iteración del bucle a cada hilo de ejecución. Es decir, cada hilo se encargará de buscar los patches de un único nivel y, una vez que termine con su tarea, si queda alguna iteración/nivel pendiente de ejecutar, se le asignará. Lo más importante que logramos de esta manera es evitar que a un único hilo de ejecución pueda corresponderle el trabajo de varios niveles de refinamiento con gran carga computacional.

Aunque la paralelización descrita anteriormente es la que se ha implementado, no está exenta de inconvenientes que deberían revisarse en un trabajo futuro. Por ejemplo, sería ideal ordenar los niveles por el número de celdas que contienen, de manera que las primeras iteraciones que se ejecuten sean las más costosas computacionalmente. Así conseguiríamos que las iteraciones más largas comiencen a ejecutarse lo antes posible, y evitaríamos el posible caso en el que el hilo que ejecuta el caso más costoso haya procesado anteriormente un nivel menos costoso, lo que resultaría en una ejecución peor balanceada.

Por otra parte, el desbalanceo de la carga entre los diferentes niveles es tan marcado que, como veremos posteriormente en los logs, la ejecución típica está dominada por uno o dos hilos con una larga ejecución, mientras que los demás hilos terminan de procesar sus niveles mucho antes. Este escenario nos indica que si paralelizásemos la generación de patches dentro de cada nivel la carga podría balancearse mucho mejor, ya que todos los hilos podrían trabajar en generar los patches de un nivel con un gran número de celdas, en vez de quedar a la espera de que el hilo procesando ese nivel muy poblado domine la ejecución y mantenga en espera al resto de hilos.

La paralelización de la detección de patches dentro de un mismo nivel genera una serie de dificultades añadidas que nos han hecho descartar esta opción, al menos dentro del alcance de este trabajo. Teniendo en mente que la construcción de patches se basa en la elección de una celda y la búsqueda de todas sus celdas adyacentes, la paralelización podría consistir en la generación de varias tareas que escogiesen diferentes celdas de partida y empezasen a construir sus propios patches en paralelo. Este escenario requeriría el uso de cerrojos para proteger el acceso a las celdas, concretamente el momento en que cada una de las tareas marca las celdas como pertenecientes al patch que están generando: debemos proteger la condición de carrera en la que dos tareas accediesen a una misma celda y la marcasen como perteneciente a su patch. Además, que dos tareas alcancen la misma celda significará que los dos patches que cada una estaba construyendo deberán fusionarse en un único patch. Esta fusión de patches, que puede no parecer demasiado complicada, requerirá tomar la decisión de qué tarea fusiona los patches. Además, parece altamente probable que se dé el caso en que sean múltiples las tareas que han comenzado a construir un mismo patch desde celdas de partida diferentes. En el peor caso, estaríamos generando tareas que comenzasen por celdas próximas y pertenecientes al mismo patch, lo que supondría generar tareas que realizarían poco trabajo antes de descubrir que el patch que llevan construido hace parte de otro más grande que está siendo creado por otras tareas. Podríamos intentar disminuir la probabilidad de que esto ocurra asignando a las tareas celdas de partida que se encuentren físicamente distantes entre sí, pero este acceso a celdas distantes no es trivial con el actual diseño, aunque la utilización de *cubekeys* podría facilitar esta implementación en un futuro.

5. Resultados obtenidos

Comparación de rendimiento entre las versiones 1 y 2 con distintos inputs

Para la comparación de los resultados obtenidos hemos tenido que realizar las ejecuciones en la máquina brutus, ya que como se podrá ver, el consumo de memoria de la nueva versión se ha incrementado considerablemente. Al trabajar en brutus, debemos tener en cuenta que se trata de un único nodo que se accede de manera compartida y pueden existir otros trabajos de diferentes usuarios ejecutándose a la vez. Por ello hemos realizado varias ejecuciones de cada caso para no confiar en una única ejecución que puede verse alterada por trabajos ajenos al nuestro.

Todos los cálculos de speed-up presentados a continuación hacen referencia a la mejora de rendimiento al comparar la versión 1 frente a la versión 2. Habrá casos en los que se compararán ejecuciones con paralelización híbrida, y otros en los que únicamente se habrá utilizado la paralelización con OpenMP. Todos los casos han utilizado 8 hilos en OpenMP, y en la ejecución híbrida se han utilizado 4 tareas MPI con 8 hilos OpenMP cada una.

Tras enviar ejecuciones secuenciales y paralelas de la versión 1 y 2 mediante el sistema de colas Slurm en la máquina brutus, hemos obtenido los siguientes resultados.

5.1. CLUES

El análisis más exhaustivo del comportamiento de la aplicación lo hemos realizado con el caso CLUES (82), que como se ha explicado anteriormente (Sección 3.4.1) corresponde al proyecto centrado en simular el grupo local (83). Al no poder ejecutarse este caso con MPI debido a las limitaciones de la implementación, el clúster kahan no cuenta con suficiente memoria por nodo para realizar estas pruebas, por lo que hemos utilizado el nodo brutus.

Registro de trabajos ejecutados con Slurm en la máquina brutus

```
     User        JobID    JobName    Account  MaxVMSize      NCPUS   NTasks  AllocCPUS    Elapsed      State
--------- ------------ ---------- ---------- ---------- ---------- -------- ---------- ---------- ----------
[AHFv1 Serie]
  fcampos 25897        AHFv1_CLU+     serial                     1                1   00:58:16  COMPLETED
          25897.0      AHF-v1.0-+     serial  14079280K          1        1       1   00:58:16  COMPLETED

  fcampos 25899        AHFv1_CLU+     serial                     1                1   01:11:44  COMPLETED
          25899.0      AHF-v1.0-+     serial  14079280K          1        1       1   01:11:43  COMPLETED


[AHFv2 Serie]
  fcampos 25900        AHFv2_CLU+     serial                     1                1   00:51:21  COMPLETED
          25900.0      AHF-v2.0-+     serial  30876840K          1        1       1   00:51:21  COMPLETED

  fcampos 25901        AHFv2_CLU+     serial                     1                1   01:00:42  COMPLETED
          25901.0      AHF-v2.0-+     serial  30876840K          1        1       1   01:00:42  COMPLETED


[AHFv1 paralela]
  fcampos 25998        AHFv1_CLU+    16cores                     8                8   00:45:32  COMPLETED
          25998.0      AHF-v1.0-+    16cores  17752480K          8        1       8   00:45:32  COMPLETED

  fcampos 26001        AHFv1_CLU+    16cores                     8                8   00:44:11  COMPLETED
          26001.0      AHF-v1.0-+    16cores  18103196K          8        1       8   00:44:11  COMPLETED


[AHFv2 paralela]
  fcampos 25949        AHFv2_CLU+    16cores                     8                8   00:35:47  COMPLETED
          25949.0      AHF-v2.0-+    16cores  32767472K          8        1       8   00:35:47  COMPLETED

  fcampos 25999        AHFv2_CLU+    16cores                     8                8   00:33:05  COMPLETED
          25999.0      AHF-v2.0-+    16cores  32767280K          8        1       8   00:33:05  COMPLETED

  fcampos 26000        AHFv2_CLU+    16cores                     8                8   00:30:24  COMPLETED
          26000.0      AHF-v2.0-+    16cores  31771260K          8        1       8   00:30:24  COMPLETED
```

Comparando las ejecuciones serie, podemos apreciar que la versión original requiere entre 58 y 71 minutos, mientras la versión 2 requiere entre 51 y 60 minutos, lo que nos permite observar una pequeña mejora de rendimiento al utilizar las nuevas estructuras de datos y el nuevo algoritmo para la generación del árbol de patches.

En las ejecuciones paralelizadas con OpenMP hemos utilizando únicamente 8 hilos (con un core por hilo) ya que la región paralelizada tiene su escalabilidad limitada por los niveles de refinamiento alcanzados y un mal balanceo de carga debido a las diferencias en el número de celdas en cada uno de ellos. Apreciamos que la versión 1 requiere un tiempo de ejecución de entre 44 y 45 minutos. En cambio, la versión 2 ejecuta entre 30 y 35 minutos, logrando una reducción de aproximadamente 10 minutos.

Tomando el mejor caso de la versión original (44 minutos) y el peor caso de la nueva (35 minutos), estaremos alcanzando un speed-up global de la aplicación de 1,25. Si tomásemos el peor caso de la versión 1 (45 minutos) y el mejor de la versión 2 (30 minutos), el speed-up sería de 1,5. Debemos tener en cuenta que estas cifras corresponden a una ejecución no distribuida con MPI, por lo que en una ejecución en la que se pueda utilizar esta distribución inter-nodos, el ahorro total en horas de CPU puede ser considerable.

Revisando el comportamiento de los hilos de ejecución que generan los patches, podemos observar en el log cómo éstos se invocan simultáneamente para los distintos niveles de refinamiento.

Fragmento de log mostrando distribución de trabajo entre hilos

```
[patch.c:623] Launching OpenMP threads in for-loop calling patches_generation_per_level().
        (thread 0/8) Calling patches_generation_per_level(level= 9)
        (thread 4/8) Calling patches_generation_per_level(level=11)
        (thread 5/8) Calling patches_generation_per_level(level=10)
        (thread 1/8) Calling patches_generation_per_level(level=15)
        (thread 6/8) Calling patches_generation_per_level(level=12)
        (thread 2/8) Calling patches_generation_per_level(level=13)
        (thread 3/8) Calling patches_generation_per_level(level=14)
        (thread 7/8) Calling patches_generation_per_level(level=16)
        (thread 0/8) We built 5329 patches (614111 rejected) at level 9 in 10.425093 seconds
        (thread 0/8) Calling patches_generation_per_level(level=17)
        (thread 5/8) We built 3357 patches (19183 rejected) at level 10 in 13.094524 seconds
        (thread 5/8) Calling patches_generation_per_level(level=18)
        (thread 5/8) We built 7805 patches (76916 rejected) at level 18 in 16.134773 seconds
        (thread 5/8) Calling patches_generation_per_level(level=19)
        (thread 4/8) We built 3462 patches (25626 rejected) at level 11 in 33.564326 seconds
        (thread 4/8) Calling patches_generation_per_level(level=20)
        (thread 0/8) We built 19955 patches (160992 rejected) at level 17 in 24.735208 seconds
        (thread 0/8) Calling patches_generation_per_level(level=21)
        (thread 0/8) We built 294 patches (14043 rejected) at level 21 in 3.067519 seconds
        (thread 0/8) Calling patches_generation_per_level(level=22)
        (thread 4/8) We built 576 patches (22605 rejected) at level 20 in 5.093121 seconds
        (thread 4/8) Calling patches_generation_per_level(level=23)
        (thread 4/8) We built 152 patches (5649 rejected) at level 23 in 0.585068 seconds
        (thread 4/8) Calling patches_generation_per_level(level=24)
        (thread 4/8) We built 6 patches (1071 rejected) at level 24 in 0.017435 seconds
        (thread 4/8) Calling patches_generation_per_level(level=25)
        (thread 4/8) We built 0 patches (6 rejected) at level 25 in 0.000063 seconds
        (thread 0/8) We built 162 patches (8503 rejected) at level 22 in 1.943257 seconds
        (thread 5/8) We built 1141 patches (35027 rejected) at level 19 in 11.154415 seconds
        (thread 7/8) We built 32168 patches (272269 rejected) at level 16 in 59.547644 seconds
        (thread 1/8) We built 41823 patches (401283 rejected) at level 15 in 130.553722 seconds
        (thread 3/8) We built 51587 patches (587216 rejected) at level 14 in 149.203084 seconds
        (thread 6/8) We built 7817 patches (86602 rejected) at level 12 in 149.381881 seconds
        (thread 2/8) We built 31029 patches (365017 rejected) at level 13 in 211.240604 seconds
(thread 0/1) All (and ONLY) the calls to patches_generation_per_level() took 211.243558 seconds
```

En este caso, la suma de los tiempos en cada nivel es de 819 segundos, pero gracias a la paralelización intra-nodo con OpenMP hemos completado todas las tareas en 211 segundos logrando un speed-up de 3,88 en la región de generación de patches.

Revisando el uso de memoria que requieren las distintas versiones ejecutadas, se puede observar cómo la nueva versión duplica las necesidades de memoria. La versión secuencial pasa de 14 a 30GB y la versión paralela de 18 a 32GB. Como ya se comentó anteriormente, durante la descripción de la implementación de tablas hash *uthash*, cada una de las estructuras almacenadas requiere la inclusión de un *handler* que ocupa 56 bytes. Observando el gran número de celdas que tienen la mayoría de los niveles de refinamiento, podemos intuir que la sobrecarga debida a la implementación puede ser muy significativa, aunque no es el único factor que aumenta los requerimientos de memoria. En concreto, sumando los valores de *HASH_COUNT* del fragmento de log, podemos observar que el número total de celdas es 97.154.443. Multiplicando por los 56 bytes, la sobrecarga alcanza los 5.440.648.808 bytes, cerca de 5GB.

La implementación de tablas *uthash* provee una rutina que devuelve la sobrecarga de memoria total. Se debe tener en cuenta que, además del *handler* insertado en la estructura, la tabla debe mantener internamente sus *buckets*, punteros y contadores. Esta información se encuentra también en el log de ejecución.

Fragmento de log mostrando sobrecarga de tablas hash

```
Freeing hash tables of subcubes, and arrays of pointers to subcubes from patches
 -Table at level= 9: COUNT=   1626277 subcubes ( 156122592 bytes), OVERHEAD= 107848792 bytes [TOTAL=  263971384]
 -Table at level=10: COUNT=   1662901 subcubes ( 159638496 bytes), OVERHEAD= 109899736 bytes [TOTAL=  269538232]
 -Table at level=11: COUNT=   5868816 subcubes ( 563406336 bytes), OVERHEAD= 395762624 bytes [TOTAL=  959168960]
 -Table at level=12: COUNT=  17764991 subcubes (1705439136 bytes), OVERHEAD=1263275016 bytes [TOTAL= 2968714152]
 -Table at level=13: COUNT=  28229226 subcubes (2710005696 bytes), OVERHEAD=2117707632 bytes [TOTAL= 4827713328]
 -Table at level=14: COUNT=  18363211 subcubes (1762868256 bytes), OVERHEAD=1296775336 bytes [TOTAL= 3059643592]
 -Table at level=15: COUNT=  10748201 subcubes (1031827296 bytes), OVERHEAD= 736117048 bytes [TOTAL= 1767944344]
 -Table at level=16: COUNT=   7026624 subcubes ( 674555904 bytes), OVERHEAD= 527708736 bytes [TOTAL= 1202264640]
 -Table at level=17: COUNT=   3832228 subcubes ( 367893888 bytes), OVERHEAD= 248159264 bytes [TOTAL=  616053152]
 -Table at level=18: COUNT=   2139373 subcubes ( 205379808 bytes), OVERHEAD= 153359384 bytes [TOTAL=  358739192]
 -Table at level=19: COUNT=   1416720 subcubes ( 136005120 bytes), OVERHEAD=  96113600 bytes [TOTAL=  232118720]
 -Table at level=20: COUNT=    849655 subcubes (  81566880 bytes), OVERHEAD=  55969352 bytes [TOTAL=  137536232]
 -Table at level=21: COUNT=    503009 subcubes (  48288864 bytes), OVERHEAD=  32362872 bytes [TOTAL=   80651736]
 -Table at level=22: COUNT=    301929 subcubes (  28985184 bytes), OVERHEAD=  19005240 bytes [TOTAL=   47990424]
 -Table at level=23: COUNT=     96259 subcubes (   9240864 bytes), OVERHEAD=   6439144 bytes [TOTAL=   15680008]
 -Table at level=24: COUNT=      4189 subcubes (    402144 bytes), OVERHEAD=    267416 bytes [TOTAL=     669560]
 -Table at level=25: COUNT=        12 subcubes (      1152 bytes), OVERHEAD=      1248 bytes [TOTAL=       2400]
        =TIMING= Freeing memory of UThash tables of subcubes and internal UTarrays of particles took 52.727563 seconds
```

Sumando el *overhead* de cada una de las tablas obtenemos un total de 7.166.772.440 bytes, 7 GB aproximadamente, mientras que la suma de toda la memoria ocupada por las celdas (*subcubes*) en cada nivel supone un total de 9.641.627.616 bytes, unos 9.6 GB. Con esta relación entre la sobrecarga y el consumo de memoria de las celdas, parece que hemos identificado el origen de gran parte del sobrecoste y que una implementación de tablas hash más eficiente reduciría considerablemente el consumo de memoria. Este sobrecoste de memoria debido a la estructura de datos es independiente de si utilizamos la versión serie o paralela.

De manera análoga al perfilado que realizamos para el caso de *Haloes going MAD* (Sección 3.5), hemos generado trazas de ejecución para la versión 1 y 2, paralelizada con OpenMP. Mostramos la visualización de las mismas con Paraver en las figuras 17 y 18.

En la figura 17 podemos apreciar el reparto de trabajo entre hilos OpenMP correspondiente a la paralelización de la generación de patches en las primeras barras de color fucsia de diferentes longitudes, debido al pobre balanceo de la carga entre niveles con grandes diferencias en el número de celdas que albergan. Después observamos una única barra en el penúltimo hilo de ejecución que indica la ausencia de reparto de trabajo entre los hilos y que se corresponde con el cálculo de parámetros físicos citado anteriormente. Por último, casi al final de la ejecución, encontramos una distribución de trabajo bien balanceada correspondiente a la generación de los halos a partir del árbol de patches generado. A pesar de que este código no ha sido desarrollado como parte de este trabajo, sino por parte de Alexander Knebe como adaptación a la nueva versión de la identificación de halos a partir de los patches, parece que las nuevas estructuras han permitido una implementación paralelizable. También es destacable cómo la sección paralelizada con OpenMP al final de la ejecución en la versión 1 se vuelve imperceptible en una ejecución de mayor tamaño, comparado con el análisis de la sección 3.5.

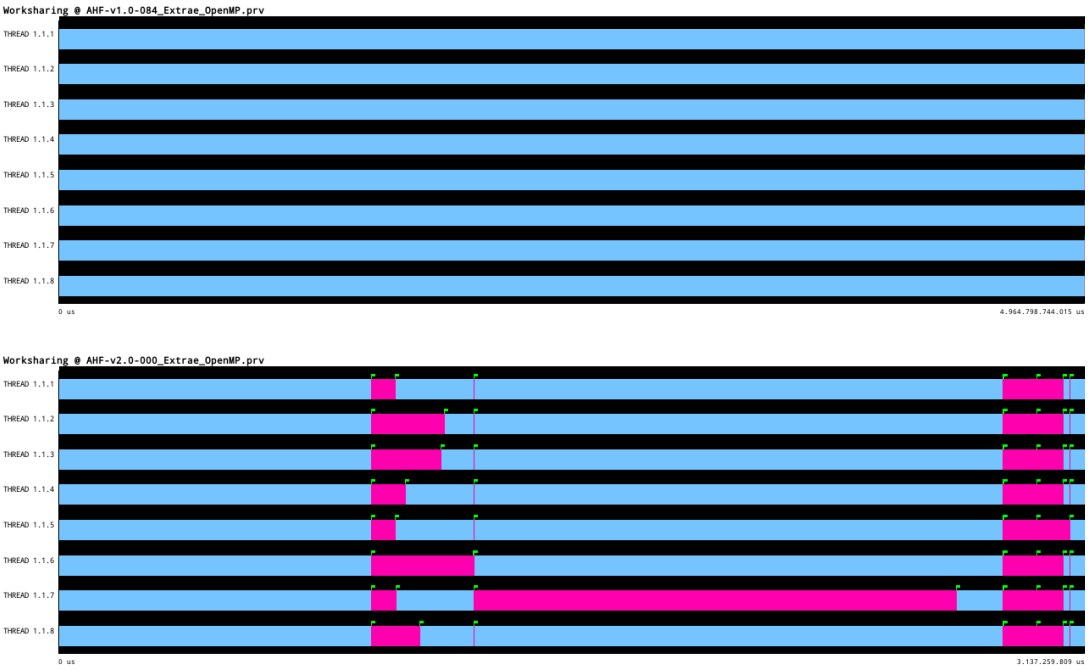

Figura 17. Reparto de trabajo entre hilos OpenMP, en fucsia las regiones que ejecutan regiones paralelas. Arriba AHFv1, abajo AHFv2

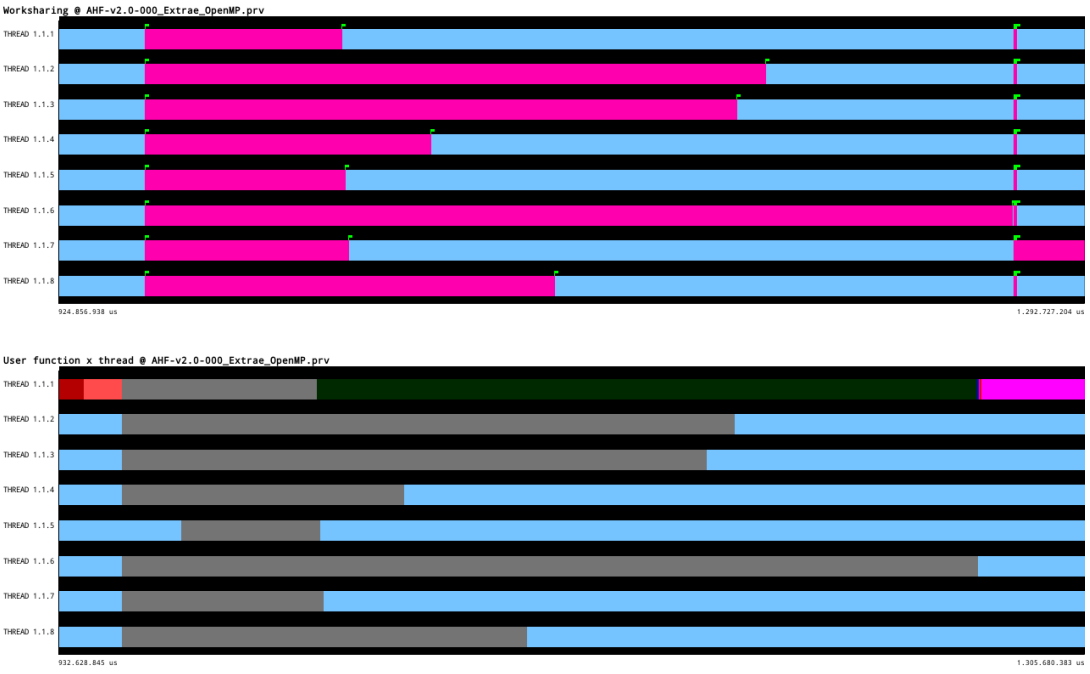

Figura 18. Reparto de trabajo y funciones de usuario, AHFv2. Zoom en región de generación de patches.

En la figura 18 se muestra un zoom de la región de generación de patches y una comparación entre las zonas de ejecución paralela y las funciones de generación de patches en cada nivel (barras grises de la imagen inferior).

Con las indicaciones del desarrollador principal de la aplicación, hemos realizado la comprobación de los resultados físicos obtenidos mediante la función de masa acumulada de los halos encontrados. Esta función, partiendo del fichero de halos ordenado de manera descendente por masa, consiste en la representación de cada uno de los objetos con una cruz ubicada en la posición que indique su masa (eje x). Al comparar los resultados obtenidos con la versión original y nueva de AHF obtenemos la figura 19.

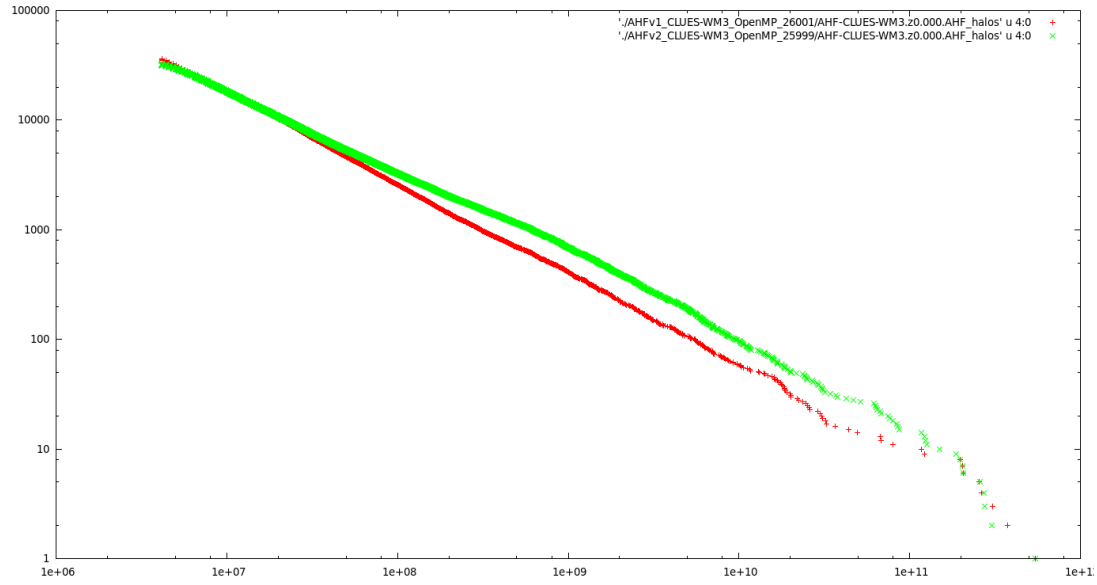

Figura 19. Función de masa acumulada para los objetos encontrados con AHFv1 y AHFv2

Puede observarse que el número total de objetos encontrados por AHFv1 es mayor: las cruces rojas en la zona superior izquierda alcanzan mayor altura en el eje y, correspondiente al número de objeto representado. Por otra parte, al observar cómo la mayoría de objetos verdes, correspondientes a los resultados de AHFv2, se encuentran más a la derecha que los rojos, podemos observar que los objetos detectados por la nueva versión son más masivos.

5.2. Haloes Going MAD

Hemos utilizado este test para hacer una prueba de ejecución híbrida (MPI+OpenMP) con la simulación de 512^3 partículas, utilizando 4 tareas MPI y 8 hilos (1 hilo por core) por tarea. La prueba, debido al alto consumo de memoria, la hemos tenido que realizar en brutus, ejecutando todas las tareas MPI en el mismo nodo. Por esta razón, los retardos debidos a la comunicación de la descomposición de dominio serán menores al no necesitar el uso de una red de interconexión, pero esta ventaja será igual tanto para la versión original como para la nueva. AHFv1 necesitó entre 52 y 53 minutos para ejecutarse, mientras que AHFv2 terminó el

post-procesado en 23 minutos en las dos ejecuciones realizadas, lo cual supone un speed-up global de 2,3.

La simulación de 256^3 partículas la hemos procesado con la versión OpenMP sin MPI, pasando de 9 minutos de ejecución para la versión 1 a 6 minutos con la versión 2 (speed-up de 1,5).

Al realizar las pruebas con la simulación de 128^3 partículas, los tiempos son tan breves (entre 15 y 19 segundos, independientemente de la versión utilizada), que la modificación realizada no aporta ninguna mejora. Es probable que el tiempo de ejecución esté dominado más por los retardos en las llamadas al sistema para la gestión de memoria, y por los tiempos de lectura y escritura de ficheros, que por el cómputo que hemos buscado optimizar y paralelizar en este trabajo.

5.3. Subhaloes Going Notts

El caso de test *Subhaloes Going Notts*, procedente del workshop del mismo nombre citado en la sección 3.4.1, lo hemos ejecutado igualmente con OpenMP y 8 hilos de ejecución. Hemos observado una reducción del tiempo de ejecución desde los entre 4 y 5 minutos de ejecución de la versión 1 a unos 2 minutos y medio, representando un speed-up de entre 1,8 y 2.

5.4. Resumen de resultados

Mediante la siguiente tabla representamos los distintos casos de input con los que hemos comparado las dos versiones de AHF, así como las características de ejecución y de la simulación analizada. Los tiempos de ejecución están expresados en minutos.

Simulación	Paralelización	Nº partículas	AHFv1	AHFv2	Speed-Up
CLUES	Serie	120.980.808	58-71	51-60	1 - 1,39
	OpenMP	120.980.808	45	33	1,36
Haloes Going MAD	MPI + OpenMP	134.217.728	52	23	2,26
	OpenMP	16.777.216	9	6	1,50
	OpenMP	2.097.152	-	-	-
Subhaloes Going Notts	OpenMP	19.170.765	4	2,5	1,6

6. Conclusiones y propuestas

Qué hemos logrado y qué podríamos lograr

Tras varios meses de trabajo hemos adquirido un conocimiento considerable sobre el problema que aborda el código AHF, así como sobre los retos intrínsecos a la implementación de sus algoritmos.

Es destacable la dificultad que implica la paralelización de este tipo de códigos en el que los datos que deberán procesarse evolucionan según se avanza en el refinamiento. A diferencia de lo que ocurre con los códigos que realizan operaciones algebraicas sobre vectores y matrices regulares, los cuales suelen estudiarse como ejemplos de paralelización, en el problema abordado en este trabajo, aun trabajando con simulaciones con un número de partículas similar y proporcionando parámetros de refinamiento idénticos, el comportamiento de la aplicación puede ser completamente diferente en función de la distribución de las partículas en la caja de simulación.

A pesar de lo anterior, hemos logrado reducir el tiempo de ejecución del código, en algunos casos de manera considerable (ver tabla sección 5.4), rediseñando las estructuras de datos empleadas y paralelizando mediante OpenMP. Teniendo en cuenta que, en las ejecuciones sobre grandes simulaciones cosmológicas, el consumo computacional del post-procesado con AHF requiere decenas de miles de horas de CPU, la mejora puede ser muy relevante. Por ejemplo, para la simulación *Big Jubilee* que cuenta con 216 miles de millones de partículas (103), el post-procesado de cada *snapshot* consumió aproximadamente 20.000 horas de CPU.

Aunque los resultados obtenidos son muy satisfactorios, aún son muchas las posibles mejoras que podrían aplicarse como trabajo futuro. Algunas de estas propuestas se han reflejado a lo largo de la presente memoria. Las más relevantes serían las siguientes.

- Desarrollar una nueva implementación de tablas hash más eficiente en cuanto a consumo de memoria.
- Modificar la función hash de las tablas, de manera que, basándonos en la *cubekey*, favorezcamos la localidad tratando de ubicar las celdas físicamente próximas en los mismos *buckets*.
- Revisar las zonas en las que se realizan numerosas llamadas a la función *realloc* y hacer reservas de memoria por bloques. Aunque el tamaño de los bloques no sea muy grande, por ejemplo 10, estaremos reduciendo el número de llamadas por 10.
- En todas las funciones que se utilizan para el cálculo del *cubekey* a partir de las coordenadas, y viceversa, y en aquellas que se utilizan para calcular las claves de las celdas adyacentes a una celda dada, podría valorarse la sustitución de bucles *for* y de saltos condicionales *if* por asignaciones de 0 o 1 a los bits en las posiciones correspondientes. Aunque el código resultante fuese más largo podría ser más eficiente al evitar las comparaciones y los saltos.

- La estructura *ahf2_patches* implementa un diseño de "estructura de arrays" (SOA), en la que se almacena un array de patches, un contador de patches almacenados y un contador de patches rechazados para cada uno de los niveles de refinamiento. Estos contadores se almacenan consecutivamente en memoria, por lo que en la misma línea de cache podremos estar almacenando los contadores de distintos niveles, accedidos por distintos hilos de ejecución. Este diseño puede estar provocando lo que se conoce comúnmente como "false sharing" (104). Aunque el acceso a los contadores no es tan intensivo, ya que entre diferentes accesos se debe procesar la detección de nuevos patches, si optásemos por utilizar un "array de estructuras" (AOS), de manera que cada estructura contuviese un único array de patches, un contador para patches almacenados y otro para los rechazados, podríamos evitar este posible conflicto (105).
- Una vez paralelizada la generación de los patches, la zona que domina el coste computacional ha podido trasladarse a otras regiones. Aunque no fuese así, puede merecer la pena valorar la paralelización de la división inicial de la caja o el refinamiento de la misma, ya que estas regiones, para simulaciones grandes, representan costes relevantes.
- Habiendo detectado que el considerable aumento de consumo de memoria puede acarrear problemas, aunque en gran parte mitigables con una mejor implementación de las tablas hash, podríamos también optar por realizar la liberación de memoria tan pronto como se terminen de procesar los datos. En concreto, las celdas serán necesarias para la identificación de patches en cada nivel, pero una vez que hayamos finalizado la identificación en cada nivel podríamos proceder a liberar su memoria y la de la tabla hash que los almacena. Actualmente esperamos a finalizar la detección de patches en todos los niveles para liberar las celdas de todas las tablas. Además podríamos lograr solapar la carga computacional de la búsqueda de patches en un nivel con la liberación de memoria de las celdas de otro nivel. Los hilos que liberarían la memoria, actualmente están a la espera de que finalice el trabajo en el nivel más poblado.

Por otra parte, teniendo en cuenta las limitaciones de la actual distribución de trabajo entre distintas tareas MPI, se nos ocurre la siguiente alternativa que requeriría un gran trabajo de rediseño y desarrollo de código.

- Como sustitución a la descomposición de dominio basada en la proximidad de las partículas (representada por sus valores en la *space filling curve*), podríamos comenzar con la lectura de los ficheros de *snapshot* cargando en memoria únicamente las coordenadas de las partículas. Procederíamos a realizar la división inicial para el nivel más grueso de refinamiento, determinado por los parámetros físicos de la simulación, y a detectar los patches en ese nivel grueso. Cada uno de los patches en este nivel tendrán un gran número de partículas al no haberse realizado el refinamiento en niveles más finos. Estos patches podrían distribuirse entre las distintas tareas MPI que realizarían el refinamiento de sus celdas y detectarían sus sub-estructuras. De esta manera la descomposición de dominio se realizaría basándonos en un refinamiento

inicial de grano grueso, en vez de en conjuntos de partículas próximas sin tener en cuenta las estructuras que forman, como se hace actualmente,.

- La división inicial descrita constituye *a priori* un procesamiento con posibilidades de ser ejecutado en aceleradores hardware ya que, a partir de las 3 coordenadas de cada partícula se calculará su *cubekey* para un nivel de refinamiento dado. Este procesado es idéntico para todas las partículas y se tendrá que realizar para cada una de ellas.

Esta propuesta estará muy condicionada por el formato de los ficheros de *snapshot* y la posibilidad de acceder posteriormente de manera directa a las propiedades físicas de las partículas.

Bibliografía

Nota: Todas las referencias web han sido accedidas en Junio de 2014

1. **Wikipedia.org.** Stonehenge. [En línea] http://en.wikipedia.org/wiki/Stonehenge.

2. —. Geocentric model. [En línea] http://en.wikipedia.org/wiki/Geocentrism.

3. —. Heliocentrism. [En línea] http://en.wikipedia.org/wiki/Heliocentric.

4. —. Newton's law of universal gravitation. [En línea] http://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation.

5. —. Espectroscopía astronómica. [En línea] http://es.wikipedia.org/wiki/Espectroscopia_astron%C3%B3mica.

6. —. Redshift. [En línea] http://en.wikipedia.org/wiki/Redshift.

7. **Nature.** Waves from the Big Bang. [En línea] http://www.nature.com/news/b-mode-1.14884.

8. **BICEP2 Collaboration.** BICEP2 2014 Release Frequently Asked Questions. [En línea] http://bicepkeck.org/faq.html.

9. **P. A. R. Ade et al. (BICEP2 Collaboration).** Detection of B-Mode Polarization at Degree Angular Scales by BICEP2. [En línea] Junio de 2014. http://journals.aps.org/prl/abstract/10.1103/PhysRevLett.112.241101.

10. **CERN.** The Large Hadron Collider. [En línea] http://home.web.cern.ch/topics/large-hadron-collider.

11. **University of Wisconsin-Madison, National Science Foundation.** IceCube, South Pole neutrino observatory. [En línea] http://icecube.wisc.edu/.

12. **CFHTLenS (The Canada France Hawaii Telescope Lensing Survey).** What is Gravitational Lensing? [En línea] http://www.cfhtlens.org/public/what-gravitational-lensing.

13. **Wikipedia.org.** Bullet Cluster. [En línea] http://en.wikipedia.org/wiki/Bullet_Cluster.

14. **NASA.** NASA Finds Direct Proof of Dark Matter. [En línea] Agosto de 2006. http://www.nasa.gov/home/hqnews/2006/aug/HQ_06297_CHANDRA_Dark_Matter.html.

15. **Gerard Lemson & the Virgo Consortium.** Millenium Run. [En línea] http://www.mpa-garching.mpg.de/millennium/.

16. **Anatoly Klypin, Joel Primack et al.** The Bolshoi Simulation. [En línea] http://hipacc.ucsc.edu/Bolshoi/index.html.

17. **Proyecto Multidark.** MultiDark simulation database. [En línea] http://www.multidark.org/MultiDark/Help?page=databases/mdr1/database.

18. **eScience Group, Leibniz-Institute for Astrophysics Potsdam (AIP).** MultiDark project. [En línea] http://www.cosmosim.org/cms/simulations/multidark-project/.

19. —. Simulations overview. [En línea] http://www.cosmosim.org/cms/simulations/simulations-overview/.

20. **Wikipedia.org.** N-body simulation. [En línea] https://en.wikipedia.org/wiki/N-body_simulation.

21. **Michele Trenti and Piet Hut (2008), Scholarpedia.** N-body simulations (gravitational). [En línea] www.scholarpedia.org/article/N-body_simulations.

22. **Wikipedia.org.** Parsec. [En línea] http://en.wikipedia.org/wiki/Parsec.

23. **NASA.** Wilkinson Microwave Anisotropy Probe. [En línea] http://map.gsfc.nasa.gov/.

24. **ESA and the Planck Collaboration.** Planck project overview. [En línea] http://www.esa.int/Our_Activities/Space_Science/Planck_overview.

25. **Michele Trenti and Piet Hut (2008), Scholarpedia.** Mean field approach: the Boltzmann equation, N-body simulations (gravitational). [En línea] http://www.scholarpedia.org/article/N-body_simulations#Mean_field_approach:_the_Boltzmann_equation.

26. **Barnes, Josh y Hut, Piet.** A hierarchical O(N log N) force-calculation algorithm. [En línea] 1986. http://www.nature.com/nature/journal/v324/n6096/abs/324446a0.html.

27. **Wikipedia.org.** Barnes–Hut simulation. [En línea] http://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation.

28. —. Particle Mesh. [En línea] http://en.wikipedia.org/wiki/Particle_Mesh.

29. —. P3M. [En línea] http://en.wikipedia.org/wiki/P3M.

30. **Springel, Volker.** GADGET-2. A code for cosmological simulations of structure formation. [En línea] http://www.mpa-garching.mpg.de/gadget/.

31. **Wikipedia.org.** Smoothed-Particle Hydrodynamics. [En línea] http://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics.

32. **Free Software Foundation, GNU project.** GSL - GNU Scientific Library. [En línea] http://www.gnu.org/software/gsl/.

33. **Matteo Frigo, Steven G. Johnson.** FFTW, "Fastest Fourier Transform in the West". [En línea] http://www.fftw.org/.

34. **HDF5 group (NCSA, LLNL, SNL, LANL).** HDF5. [En línea] http://www.hdfgroup.org/HDF5/.

35. **eScience Group, Leibniz-Institute for Astrophysics Potsdam (AIP).** Halo Finders. [En línea] http://www.cosmosim.org/cms/simulations/halo-finders/.

36. **Alexander Knebe et al.** AMIGA halo finder (AHF). [En línea] http://popia.ft.uam.es/AHF.

37. **Wikipedia.org.** Hot Spot (computer programming). [En línea] http://en.wikipedia.org/wiki/Hot_spot_%28computer_science%29.

38. —. Bottleneck. [En línea] http://en.wikipedia.org/wiki/Bottleneck.

39. —. Hardware counter. [En línea] http://en.wikipedia.org/wiki/Hardware_counter.

40. **KDE - K Desktop Environment incorporated society (KDE e.V.) .** Profiling Methods. *Kcachegrind Documentation.* [En línea] http://docs.kde.org/development/en/kdesdk/kcachegrind/introduction-methods.html.

41. **Wikipedia.org.** Statistical Profilers. [En línea] http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29#Statistical_profilers.

42. —. Instrumentation. Profiling (computer programming). [En línea] http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29#Instrumentation.

43. **OpenMP Architecture Review Board.** OpenMP Application Program Interface. Version 4.0. *Sección 3.4.1, pág 233.* [En línea] Julio de 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

44. **Open MPI project.** MPI_Wtime. *man page (version 1.5.5).* [En línea] https://www.open-mpi.org/doc/v1.5/man3/MPI_Wtime.3.php.

45. **University of Oregon.** TAU User Guide. *Chapter 1. Tau Instrumentation.* [En línea] http://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt01ch01.html#TauLibraryInterposition.

46. **BSC-CNS. Computer Science, Performance Tools group.** Extrae. *Linker preload (LD_PRELOAD).* [En línea] http://www.bsc.es/computer-sciences/extrae.

47. **Linux man-pages project.** ld.so/ld-linux.so - dynamic linker/loader. *ENVIRONMENT: LD_PRELOAD.* [En línea] http://man7.org/linux/man-pages/man8/ld.so.8.html.

48. **M. Tim Jones.** Anatomy of Linux dynamic libraries. [En línea] http://www.ibm.com/developerworks/library/l-dynamic-libraries/.

49. **University of Oregon.** TAU Instrumentation API. [En línea] http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03rn01.html.

50. **BSC-CNS. Computer Science, Performance Tools group.** Extrae User Guide. *Section 5, Extrae API.* [En línea] http://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide#SECTION00800000000000000000.

51. —. Extrae User Guide. *Section 5.1, Basic API.* [En línea] http://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide#SECTION00810000000000000000.

52. **University of Maryland, The University of Wisconsin-Madison.** Paradyn/Dyninst. [En línea] http://www.dyninst.org/.

53. —. Dyninst API. [En línea] http://www.dyninst.org/dyninst.

54. **University of Oregon.** Running an application using DynInstAPI. [En línea] http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch02s02.html.

55. **BSC-CNS. Computer Science, Performance Tools group.** Extrae User Guide. *Section 4.8, XML Section: User functions.* [En línea] http://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide#SECTION00780000000000000000.

56. **Wikipedia.org.** TLB, Translation lookaside buffer. [En línea] http://en.wikipedia.org/wiki/Translation_lookaside_buffer.

57. **PAPI project.** PAPI Standard Events By Architecture. [En línea] http://icl.cs.utk.edu/projects/papi/presets.html.

58. **Innovative Computing Laboratory (The University of Tennessee).** Performance Application Programming Interface (PAPI). [En línea] http://icl.cs.utk.edu/papi/index.html.

59. **BSC-CNS. Computer Science, Performance Tools group.** Paraver. General overview. [En línea] http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview.

60. **University of Oregon.** ParaProf - User's Manual. [En línea] http://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt02ch07.html.

61. —. ParaProf - User's Manual. *Supported Formats.* [En línea] http://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt02ch07s02.html.

62. **ZIH (Technische Universität Dresden), University of Oregon, LLNL.** Open Trace Format (OTF). [En línea] http://www.tu-dresden.de/zih/otf/.

63. **Score-P project.** Open Trace Format 2. [En línea] https://silc.zih.tu-dresden.de/otf2-current/html/.

64. **VI-HPS.** Score-P. *Scalable Performance Measurement Infrastructure for Parallel Codes.* [En línea] http://www.vi-hps.org/projects/score-p/.

65. **ParaTools, Inc. .** Open Trace Format. [En línea] http://www.paratools.com/OTF.

66. **Jülich Supercomputing Centre (Forschungszentrum Jülich), Laboratory for Parallel Programming (German Research School for Simulation Sciences).** Scalasca. [En línea] http://www.scalasca.org/.

67. **Center for Applied Mathematics of Research Center Jülich, Center for High Performance Computing of the Technische Universität Dresden.** Vampir. [En línea] http://www.vampir.eu/.

68. **Wikipedia.org.** Gprof. [En línea] http://en.wikipedia.org/wiki/Gprof.

69. **Free Software Foundation, GNU project.** GNU gprof manual. *binutils package documentation.* [En línea] http://sourceware.org/binutils/docs/gprof/.

70. **Knebe, Alexander.** AHF-AMIGA's Halo Finder. *User's Guide.* [En línea] http://popia.ft.uam.es/AHF/files/AHF.pdf.

71. **Gill, Stuart P. D.; Knebe, Alexander; Gibson, Brad K.** The evolution of substructure - I. A new identification method. [En línea] Junio de 2004. http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=2004MNRAS.351..399G&db_key=AST&high=4174ffa0a814894.

72. **Knollmann, Steffen R.; Knebe, Alexander.** AHF: Amiga's Halo Finder. [En línea] Junio de 2009. http://adsabs.harvard.edu/abs/2009ApJS..182..608K.

73. **Alexander Knebe, Andrew Green, James Binney.** Multi-level adaptive particle mesh (MLAPM): a c code for cosmological simulations. [En línea] Agosto de 2001. http://mnras.oxfordjournals.org/content/325/2/845.short.

74. **Alexander Knebe et al.** Structure finding in cosmological simulations: the state of affairs. [En línea] Octubre de 2013. http://adsabs.harvard.edu/abs/2013MNRAS.435.1618K.

75. —. Structure finding in cosmological simulations: the state of affairs. [En línea] Julio de 2013. http://arxiv.org/abs/1304.0585.

76. **Wikipedia.org.** Hilbert curve. [En línea] http://en.wikipedia.org/wiki/Hilbert_curve.

77. —. Peano curve. [En línea] http://en.wikipedia.org/wiki/Peano_curve.

78. **Knebe, Alexander, y otros.** Haloes Going MAD workshop. [En línea] Mayo de 2010. http://popia.ft.uam.es/HaloesGoingMAD/Home.html.

79. **Knebe, Alexander et al.** Subhaloes Going Notts workshop. [En línea] Mayo de 2012. http://popia.ft.uam.es/SubhaloesGoingNotts/Home.html.

80. **Knebe, Alexander, y otros.** Haloes going MAD. *Publications and related workshops.* [En línea] http://popia.ft.uam.es/HaloesGoingMAD/Publications.html.

81. —. Haloes going MAD. *The Data.* [En línea] http://popia.ft.uam.es/HaloesGoingMAD/The_Data.html.

82. **AIP (Alemania), IPNL (Francia), RIP (Israel), NMSU (EEUU), UAM (España).** CLUES Project. [En línea] www.clues-project.org/.

83. **Wikipedia.org.** Grupo Local. [En línea] http://es.wikipedia.org/wiki/Grupo_Local.

84. **DSIC, Universitat Politècnica de València.** Clúster Kahan. [En línea] http://users.dsic.upv.es/~jroman/kahan.html.

85. **Departamento de Física Teórica. UAM.** Astrophysics and Cosmology group. [En línea] http://astro.ft.uam.es/.

86. **BSC-CNS. Computer Science, Performance Tools group.** Extrae User's Guide. *Section 4, Extrae XML configuration file.* [En línea] http://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide#SECTION00700000000000000000.

87. **Man pages.** getrusage(2) - Linux man page. [En línea] http://linux.die.net/man/2/getrusage.

88. **Wikipedia.org.** Find first set. *Hardware support.* [En línea] http://en.wikipedia.org/wiki/Find_first_set#Hardware_support.

89. —. Self-balancing binary search tree. [En línea] http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree.

90. —. B-tree. [En línea] http://en.wikipedia.org/wiki/B_tree.

91. —. Hash table. [En línea] http://en.wikipedia.org/wiki/Hash_table.

92. —. Hash function. [En línea] http://en.wikipedia.org/wiki/Hash_function.

93. **cplusplus.com.** unordered_map. [En línea] http://www.cplusplus.com/reference/unordered_map/unordered_map/.

94. **Wikipedia.org.** Standard Template Library. [En línea] http://en.wikipedia.org/wiki/Standard_Template_Library.

95. —. C++11. *Hash tables.* [En línea] http://en.wikipedia.org/wiki/C%2B%2B11#Hash_tables.

96. **Hanson, Troy D.** uthash: a hash table for C structures. [En línea] http://troydhanson.github.io/uthash/.

97. —. uthash User Guide. *A word about memory.* [En línea] http://troydhanson.github.io/uthash/userguide.html#_a_word_about_memory.

98. **The IEEE and The Open Group.** realloc. [En línea] 2004. http://pubs.opengroup.org/onlinepubs/009695399/functions/realloc.html.

99. **Hanson, Troy D.** utarray: dynamic array macros for C. [En línea] http://troydhanson.github.io/uthash/utarray.html.

100. —. utarray. *Operations.* [En línea] http://troydhanson.github.io/uthash/utarray.html#_operations.

101. **GitHub.com.** utarray Issue 21. *ssize_t not found.* [En línea] https://github.com/troydhanson/uthash/issues/21.

102. —. uthash Issue 26. *Max elements in UThash?* [En línea] https://github.com/troydhanson/uthash/issues/26.

103. **Jubilee Project.** JUropa huBbLE volumE simulation project. *Simulations.* [En línea] http://jubilee.ft.uam.es/simulations.

104. **Intel.** Avoiding and Identifying False Sharing Among Threads. [En línea] https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads.

105. —. Memory Layout Transformations. [En línea] https://software.intel.com/en-us/articles/memory-layout-transformations.