Universidad Zaragoza

Proyecto Fin de Carrera

Ingeniería en Informática

Comportamiento de un cluster heterogéneo de CPUs y GPUs para el trazado de rayos

Autor

Daniel Martínez Cucalón

Directores

Francisco José Serón Arbeloa
Juan Antonio Magallón Lacarta

Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza

2016

RESUMEN

El objetivo de este PFC es realizar la adaptación de un trazador de rayos al cálculo en paralelo sobre varias computadoras conectadas en red y estudiar su comportamiento y el rendimiento conseguido.

En el texto se describe la motivación del proyecto, que nace de simular de manera matemática el fenómeno de la iluminación y como es interpretada de forma humana por medio de la visión.

A continuación, en el capítulo 2, se hace un recorrido por el estado de la tecnología con respecto a la síntesis de imágenes por computador, con las técnicas más relevantes sobre la simulación de la iluminación y los trazadores de rayos. En el mismo capítulo, se describen las tecnologías actuales en cuanto a materia de paralelización, tanto en hardware como en el software necesario para hacerlo funcionar.

En el capítulo 3, se describe el sistema hardware concreto sobre el que se despliega el cluster de pruebas, así como las tecnologías con las que se ha desarrollado el proyecto.

En el cuarto capítulo se presenta el sistema ALEPH/FTL, que es el trazador de rayos que se ha usado para realizar este estudio, haciendo un recorrido por su estructura.

En ese mismo capítulo se encuentra la explicación de como se ha afrontado el diseño de la adaptación al cálculo en paralelo del software trazador de rayos, y se detalla la implementación de la solución adoptada. Con respecto a este punto, se añade como apéndice el código en C++ de la clase que dota al motor de estas características y que es aportado por el alumno al código del motor ALEPH/FTL.

El siguiente capítulo esta dedicado a la presentación de lo necesario para realizar las pruebas: escenas, diseño de pruebas, medidas de rendimiento...

El sexto capítulo es una continuación del anterior, ya que su contenido consiste en la presentación y análisis de las pruebas realizadas.

Por último, se encuentra un capítulo con conclusiones y se indican líneas de trabajo futuro.

Agradecimientos

A mis padres, Félix y María Carmen, por hacer todo lo que ha estado en su mano para darme las oportunidades y facilidades para formarme de la mejor manera posible, tanto como ingeniero como persona.

A Francisco José Serón, por todas las oportunidades, el apoyo, la ayuda aportada en estos últimos años de la carrera y por haber confiado en mi para este PFC y otros proyectos. También por haberme hecho ver, desde la primera vez que fui su alumno, que el mundo universitario no es tan frío y que la excelencia no está reñida con el carácter humano y la cercanía.

A Juan Antonio Magallón, por su colaboración y dedicación en la puesta en marcha de todo el cluster, el código del trazador ALEPH, y su evolución FTL, los parches imposibles cuando no había manera de ver el problema, y por toda su ayuda para la realización técnica del proyecto.

Al ISAAC, al GIGA, y a los compañeros con los que he compartido laboratorio: Carlos, Tomás, Manuel, David, Eduardo...; a los otros que aún no habiendo estado en el mismo laboratorio han sido grandes compañeros durante tantos años de carrera: Andrés, Adrián, Javier, Jaime, Jorge... y de igual manera a los demás que sin haber compartido tanto tiempo también me han apoyado en momentos puntuales.

A mis amigos: Cristina, Jorge, Ana Pilar, María, y mi hermana Beatriz, por ayudarme a superar los momentos difíciles, tanto en el marco académico como en el personal.

A Jessica, por el aguante y la paciencia que tiene cuando hablo de ordenadores y demás cacharros raros; pero sobre todo por su amistad, su apoyo, su cariño y ayuda en el momento crítico... y resumiendo, por todos los momentos que hemos compartido desde hace tanto tiempo.

GRACIAS a todos, con mayúsculas.

Índice General

Índice de ilustraciones

Índice de tablas

MEMORIA

1 Introducción

1.1 Alcance del documento

El presente documento describe el trabajo realizado por el alumno Daniel Martínez Cucalón como Proyecto de Fin de Carrera, titulado “Comportamiento de un cluster heterogéneo de CPUs y GPUs para el trazado de rayos”.

Este PFC consiste en la paralelización de un sistema de trazado de rayos, su despliegue en un cluster de varias computadoras y el análisis de su comportamiento.

1.2 Contexto de desarrollo

El Grupo de Informática Gráfica Avanzada (GIGA) inició su andadura en la Universidad de Zaragoza a principios de los 90, desde entonces ha sido coordinado por el Dr. Francisco José Serón, en estos momentos profesor Catedrático de Universidad. El grupo pertenece al Instituto de Investigación de Ingeniería de Aragón,y está considerado como grupo consolidado por el Gobierno de Aragón.

Su enfoque inicial se centró en la realización de actividades de I+D+i en las áreas típicas de la Informática Gráfica tradicional. En el momento actual el grupo está formado por profesores de universidad estables alrededor de los cuales se aglutinan varios doctorandos y colaboradores del grupo cuyo número fluctúa a lo largo del tiempo en función de la financiación de que se dispone en cada momento.

El grupo tiene experiencia probada en la realización de proyectos mediante convocatoria pública competitiva a nivel regional, nacional e internacional, y ha realizado transferencia tecnológica a numerosas empresas e instituciones.

La simulación de la interacción de la luz con los materiales para generar imágenes por ordenador es una de esas áreas típicas en las que el grupo históricamente ha estado trabajando.

1.3 Motivación del proyecto

Desde hace algún tiempo se viene usando la tecnología informática para simular aspectos y comportamientos físicos del mundo natural, basándose en modelos matemáticos extraídos del ámbito científico.

Uno de estos aspectos es la interacción de los rayos de luz sobre los materiales y la percepción humana que tenemos de este fenómeno a través de la visión.

En el mundo natural, existen fuentes de luz que la irradian en todas las direcciones sobre los materiales, los cuales, al poseer distintas propiedades, interaccionan con esa radiación

bien absorbiéndola calentándose, o bien reflejándola o transmitiéndola, pudiendo a su vez cambiar sus propiedades como dirección, intensidad u otras. El hecho de reflejar esta radiación, convierte a su vez a los materiales en fuentes emisoras de luz en todas las direcciones, interactuando con los materiales del entorno de manera recursiva, y nuestros ojos, o más bien sus células receptoras (conos y bastones) al encontrarse en el camino de esa radiación, se ven estimulados por ella y nuestro cerebro lo interpreta como una imagen. Este comportamiento es el que se pretende simular mediante los trazadores de rayos.

Para comprender la magnitud del problema al que nos enfrentamos, además de tener en cuenta la recursión generada en los rebotes, sería preciso reparar también en el hecho de que la radiación puede incidir desde cualquier dirección, lo que es imposible de tratar de manera computacional, y fuerza a recurrir a la discretización de esas radiaciones y tratarlas en forma de rayos, de manera que la complejidad y la precisión de la simulación se puede ajustar de manera infinitesimal, pareciendo claro que a mayor número de rayos e interacciones sean calculadas, más preciso será el resultado, más costoso será su cálculo y por ende, más capacidad computacional será necesaria para obtener los resultados en una cantidad de tiempo menor.

La capacidad de comunicación de computadoras entre sí hace pensar en que problemas como el citado se pueden abordar en conjunto por un numero de computadoras coordinadas para resolverlos, además, estas computadoras pueden contar con diferentes características y capacidades, lo que extiende el problema a la optimización de la distribución del trabajo y la medida del rendimiento de estos sistemas heterogéneos.

1.4 Objetivos y alcance

El objetivo principal de este PFC es estudiar las posibles ventajas que aporta la capacidad del cálculo en paralelo distribuido en varias máquinas, sobre el rendimiento y los tiempos de cálculo de los software de síntesis de imágenes mediante el trazado de rayos.

Indirectamente, el código necesario para la paralelización en varias máquinas escrito para la realización de este estudio, pasa a formar parte del motor de renderizado que se ha modificado, y supone una aportación que si bien pudiera no ser definitiva, si que amplía las posibilidades del desarrollo y la evolución de este motor hacia otros aspectos y características que inicialmente no estaban contempladas.

1.5 Trabajo del autor

El trabajo del autor para estudiar el comportamiento de un cluster heterogéneo de CPUs y GPUs para el trazado de rayos ha sido el siguiente:

- Repaso de conocimientos en técnicas de programación paralela e informática gráfica adquiridas durante el transcurso de la carrera, así como otros conocimientos sobre

programación orientada a objetos, administración de sistemas y redes.

- Estudio de la implementación del motor de trazado de rayos *ALEPH* disponible en el GIGA, entendiendo su estructura y preparando conocimientos sobre las tecnologías usadas en el desarrollo de ese motor, en especial el lenguaje de programación C++11.
- Elección de las herramientas y bibliotecas a usar para la implementación de la adaptación del anterior motor al procesamiento en paralelo, así como el entrenamiento, documentación y pruebas prácticas enfocadas en la adquisición de conocimientos sobre estas.
- Implementación de la adaptación, con las distintas técnicas de comunicación entre las unidades de proceso del cluster, del motor de trazado de rayos al cálculo en paralelo.
- Diseño e implementación de pruebas para evaluar el comportamiento del sistema.
- Medición de resultados y extracción de conclusiones al respecto.

1.6 Contenido de la documentación

Este documento consta de:

- **Memoria**: que contiene los siguientes apartados:
  - **Introducción**: El presente capítulo, donde se detalla el contexto, la motivación y los objetivos del proyecto.
  - **Estado del arte**: Capítulo 2, que contiene una descripción general del estado de la tecnología con respecto a la síntesis de imagen por computador y al procesamiento de datos en paralelo.
  - **Descripción, diseño e implementación del trazador de rayos**: Capítulos 3 y 4, donde se describe la arquitectura y las características del sistema mediante el que se ha llevado a cabo este estudio. En el capítulo 4 se detalla como se ha implementado la adaptación del motor de trazado de rayos al cálculo en paralelo.
  - **Pruebas y resultados**: Capítulos 5 y 6, con descripción y resultados de las pruebas realizadas para la validación de la solución adoptada, así como el análisis de los resultados obtenidos.
  - **Conclusiones**: Capítulo final con las conclusiones y valoraciones finales.

2 Estado del Arte

2.1 Síntesis de imagen por computador

2.1.1 Proceso general: Pipeline gráfica

La síntesis de imágenes por computador es un proceso que consta de varias etapas, partiendo de la definición de la geometría, hasta llegar a la propia visualización de la imagen.

Como etapa inicial de este proceso, se lleva a cabo el modelado de la escena, definiendo la geometría de cada objeto.

Una vez definida la geometría de los objetos con respecto a unas coordenadas locales, se les aplica a estos unas transformaciones tridimensionales (posicionamiento, escala), con lo cual se obtienen las coordenadas globales del mundo, en esta etapa se posicionan las fuentes de luz, cuales poseen también unas propiedades asociadas.

Como las caras ocultas de los objetos no formarán parte de la imagen sintetizada, en este punto del proceso se eliminan de forma independiente para cada objeto.

A continuación, se procede al cálculo de la iluminación de la escena, que según los métodos empleados, se realizará de una manera o de otra, y que al ser especialmente relevante para los objetivos de este PFC se detallará en otras secciones más extensas que esta introducción.

Con la iluminación de la escena calculada, se establece un sistema de referencia visual, donde se define la posición del observador y las características ópticas de la cámara. Teniendo estas propiedades definidas, se calcula el volumen de visualización (cono de visión), y se descartan todos los polígonos, o las partes de ellos, que no están contenidos en este volumen.

Después se hace una proyección de la escena al espacio en dos dimensiones de la pantalla plana y con ello se conoce la relación de las coordenadas en el espacio plano con las globales de la escena, pertenecientes al espacio tridimensional.

Como proceso integrador de todas las etapas anteriores, se llega a la etapa de rasterización, donde se triangularizan las mallas de los objetos para asegurarse de que todos los triángulos son planos, se eliminan las superficies ocultas (cuando se encuentran otros objetos entre el plano de la imagen y las superficies), y se calcula el “color shading” para cada uno de los píxeles de la pantalla.

El color shading consiste en calcular, conociendo el modelo de iluminación y las propiedades del triángulo al que corresponde un píxel, el color para éste, así pues, las

operaciones de sombreado, coloreado y testeado, son necesarias debido a que según con que técnicas de iluminación no se calculan las intensidades para todos y cada uno de los píxeles del plano de la imagen, siendo necesaria la aplicación de tratamientos posteriores al propio cálculo de la iluminación, como pudiera ser la interpolación entre vértices, para los que sí que se ha realizado ese cálculo de manera real.

Al buscar una imagen calculada de la manera más precisa posible, para la realización de este PFC se ha enfocado en técnicas de iluminación completas en las que se calcula esta para todos y cada uno de los puntos de la imagen, no solo en los vértices de los triángulos, sin recurrir a estas técnicas de interpolación, a cambio, evidentemente, de un mayor coste computacional.

Por último, y también como parte resultante de todo el proceso descrito anteriormente, se realiza la propia visualización de la imagen.

2.1.2 Iluminación

En el proceso descrito en el punto anterior, teniendo una escena definida, y habiendo elegido un punto de vista desde el cual observarla, el problema de la síntesis de la imagen se reduce a conocer como están iluminados los puntos visibles de esta.

Así pues, uno de los aspectos que más relevancia tiene dentro del cálculo de imágenes por computador, sin tener en cuenta temas artísticos, como podría ser el propio modelado de las formas, son los modelos de iluminación, pues la simulación de esta, con todos los detalles que engloba, es la que va a dar como resultado una imagen, que será más o menos comparable a su equivalente en el mundo físico dependiendo de lo fidedigno que resulte el modelo de iluminación.

Un modelo de iluminación define analíticamente la interacción de la luz con los materiales de la escena. Para cada punto de la escena se tiene el equivalente a dos fuentes de iluminación, la directa, proveniente de las fuentes de luz, y la indirecta, que resulta de la reflejada por otros elementos de la escena.

Los materiales que componen los objetos de la escena poseen distintas propiedades, que les transfieren distintos comportamientos ópticos, como la reflexión, compuesta por las componentes especular, difusa y ambiental, que definen de que manera la luz es reflejada al llegar al material; o la refracción, que determina como cambia la dirección de la luz al transmitirse hacia el interior de un material que posea cierta transparencia. También se puede entender como una propiedad de este estilo la capacidad que algunos materiales tienen en la naturaleza, de brillar (emitir radiación lumínica por si mismo) cuando son excitados por una corriente eléctrica, calor, o radiación, y en tal caso, estos objetos pasarían a ser fuentes de luz. Este comportamiento se modela a través de la BDRF (*Bidirectional Reflectance Distribution Function*) [Hal88].

Conociendo el comportamiento de las superficies (dado por su BDRF) y los modelos matemáticos de las fuentes de luz, para llegar al objetivo de calcular la iluminación de la escena, es necesario aplicar algún modelo de transporte de luz, los cuales están basados en la ecuación integral de la radiancia.

2.1.2.1 Ecuación integral de la radiancia

El proceso de iluminación de la escena puede ser descrito con la Ecuación Integral de la Radiancia, que simplificada para tener en cuenta únicamente la reflexión toma la siguiente forma:

$$L(P,\vec{d}) = L_e(P,\vec{d}) + L_r(P,\vec{d})$$

donde:

- $L(P,\vec{d})$ es la radiancia total saliente de un punto P, en una dirección determinada $\vec{d}$.
- $L_e(P,\vec{d})$ es la radiancia emitida por la superficie desde el punto P de manera independiente al resto del entorno.
- $L_r(P,\vec{d})$ es la radiancia proveniente del resto del entorno y que es reflejada desde el punto P en la dirección $\vec{d}$.

La radiancia reflejada por la superficie depende de la radiancia total que incide sobre el punto P, saliendo desde todos los puntos de todas las superficies que componen todos los objetos de la escena; de las propiedades ópticas del material y la superficie donde está situado el punto P; y de la posición relativa de cada elemento de la escena respecto al punto P, ya que puede darse el caso de que unos puntos estén ocultos con respecto a otros. La radiancia reflejada $L_r(P,\vec{d})$ se puede calcular mediante la integral:

$$\begin{aligned} L_r(P,\vec{d}) &= \int_S \rho(P,\vec{d},\vec{d}_i)\, L_i(P,\vec{d}_i)(\vec{n}\cdot\vec{d}_i)\, dS \\ &= \int_S \rho(P,\vec{d},\vec{d}_i)\, h(P,P')\, L(P',\vec{d}_i)(\vec{n}\cdot\vec{d}_i)\, dS \end{aligned}$$

donde:

- S representa el conjunto de superficies de la escena.
- $L_i(P,\vec{d}_i)$ es la radiancia que incide sobre el punto P en una dirección determinada $\vec{d}$.
- $\rho(P,\vec{d},\vec{d}_i)$ es la reflectividad bidireccional de la superficie, la proporción de luz que llega desde la dirección $\vec{d}_i$ y es reflejada en una dirección $\vec{d}$.
- $\vec{n}$ es la normal a la superficie en el punto P.

Entonces, la ecuación integral de la radiancia puede expresarse como:

$$L(P,\vec{d})=L_e(P,\vec{d})+\int_{P'\in S}\rho(P,\vec{d},\vec{d}_i)h(P,P')L(P',\vec{d}_i)(\vec{n}\cdot\vec{d}_i)dS$$

A partir de la ecuación anterior, para cada punto de la escena es necesario determinar la integral del segundo miembro, que a su vez depende del valor de la función en otro punto del dominio. Además se debe de determinar la ocultación entre el punto P y cada uno de los puntos P' pertenecientes al dominio de integración.

Se puede utilizar como dominio de integración, en vez de las superficies de la escena, el ángulo sólido Ω_i que comprende a todas las posibles direcciones de incidencia. Normalmente este ángulo comprenderá toda la semiesfera de incidencia, o bien, si se incluyen en la ecuación los efectos de transmisión, la esfera completa. Entonces, la ecuación tomará la siguiente forma:

$$L(P,\vec{d})=L_e(P,\vec{d})+\int_{\vec{d}_i\in\Omega_i}\rho(P,\vec{d},\vec{d}_i)L(P',\vec{d}_i)(\vec{n}\cdot\vec{d}_i)d\Omega_i$$

Entonces, la variable de integración pasa a ser $\vec{d}_i$, y es necesario determinar que punto P' es visible en esa dirección.

Esta ecuación refleja todos los posibles comportamientos de la luz con la materia: difusa-difusa, difusa-especular, especular-difusa, y especular-especular.

La integración completa y exacta de la ecuación resulta inviable en tiempo necesario de cálculo y necesidades de almacenamiento, y han surgido varios algoritmos con el objetivo de abordarla basándose en introducir ciertas simplificaciones.

2.1.2.2 Trazado de rayos inverso

Esta técnica para el cálculo de la iluminación global se basa en las leyes de la óptica geométrica.

Los trazadores de rayos trabajan disparando rayos desde el punto de vista hacia la escena a través del plano de la imagen a sintetizar, que al incidir sobre las superficies de los objetos que la componen, estos se vuelven a lanzar de manera desde ese punto de incidencia hasta un nivel arbitrario de profundidad, con esto se consiguen simular los efectos de reflexión y refracción.

Los cálculos con estos métodos dependen del punto de vista del observador, ya que las trayectorias de los rayos tiene como origen este mismo punto.

Aunque el cálculo de la iluminación directa se realiza de manera similar en todos, el cálculo de la iluminación indirecta permite distinguir entre distintos métodos.

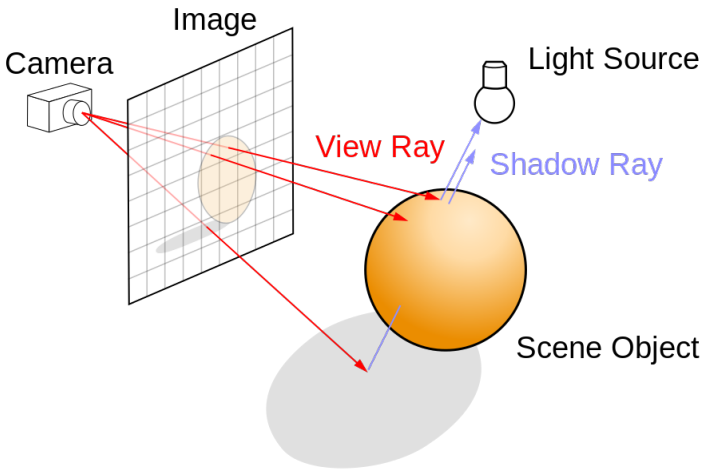


Ilustración 1: Esquema conceptual de un trazador de rayos

Trazado de rayos simple

Estos métodos únicamente tienen en cuenta uno de los comportamientos de la iluminación indirecta, el especular-especular y permiten capturar la iluminación directa especular y difusa, pero no las interacciones de tipo difuso-difuso o especular-difuso entre objetos, en los que se recibe luz de todas las direcciones.

Trazado de rayos distribuido

[CPC84] Con respecto a los anteriores, estos métodos añaden al cálculo simple el cálculo de toda la iluminación, tanto directa como indirecta, mediante la integración local de la ecuación de iluminación en un punto de la superficie.

Son muy costosos en tiempo de cálculo, ya en cada rebote de un rayo se debe evaluar una integral y por tanto, la complejidad de este cálculo crece de manera exponencial.

Métodos de Monte Carlo

Buscando reducir el elevado coste de los métodos anteriores, aparecieron métodos que tratan de resolver la ecuación integral de la radiancia de forma global a lo largo del camino de un rayo, integrando todos los rebotes en el mismo cálculo.

A este grupo pertenecen los algoritmos de path-tracing, como el usado por el motor elegido para realizar este estudio.

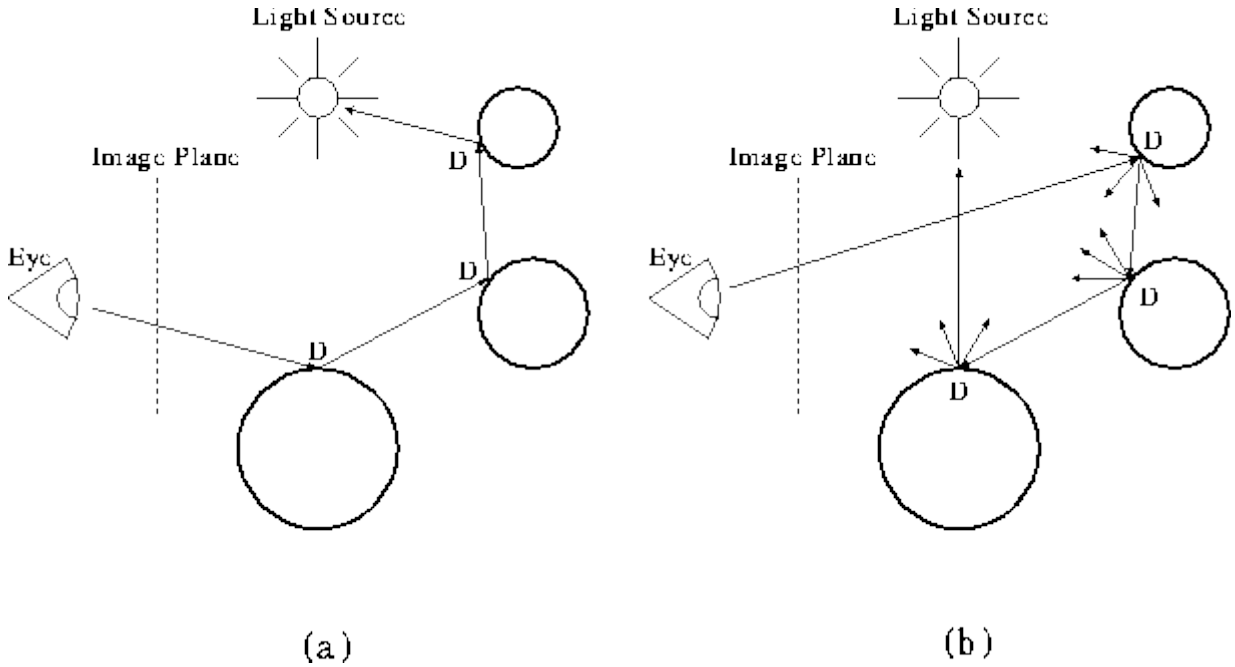


Ilustración 2: (a) Método de Monte Carlo, path-tracing. (b) Trazado de rayos distribuido.

2.1.2.3 Radiosidad

Radiosidad [CGIB86] es una técnica de iluminación global que trata de resolver el problema de la síntesis de imágenes de la forma más real, basándose en la teoría física de la transferencia de calor para simular la iluminación indirecta en escenas con superficies difusas.

La teoría de la transferencia de calor describe a la radiación como la transferencia de energía desde una superficie cuando esta ha sido excitada térmicamente. Esto incluye tanto a las superficies emisoras de energía, como las fuentes de luz, como a las que reciben energía de otras superficies y por lo tanto tienen energía para transferir.

Estos métodos, también denominados métodos de elementos de contorno, obtienen la interacción difusa-difusa y los resultados obtenidos son independientes del punto de vista del observador.

2.1.2.4 Métodos híbridos

Existen métodos de cálculo de iluminación que buscan integrar lo mejor de los dos ya descritos anteriormente, resultando en los llamados métodos híbridos o de doble pasada.

Estos métodos calculan la iluminación difusa haciendo uso de un algoritmo de radiosidad y adicionalmente se realiza una pasada de trazado de rayos simplificada, con lo que consiguen calcular las interacciones difusa-difusa y especular-especular, pero no obtienen los resultados de los otros dos modos de interacción de la luz con la materia.

2.1.2.5 Métodos directos

Otros métodos modelan la luz desde el punto de vista de la naturaleza cuántica de la luz y simular el comportamiento de los fotones que viajan desde las fuentes de luz e interaccionan con los materiales de los que se componen los objetos.

Los métodos mas representativos de este tipo son los de mapas de fotones: La iluminación se distribuye sobre los objetos en función de la cantidad de fotones que llegan a cada superficie y se almacena en los llamados mapas de fotones [Jen01].

2.1.3 Path Tracing

Los algoritmos de path-tracing, como el utilizado como base de este estudio, pueden verse como un trazador de rayos simple, al que se le han añadido capacidades de un trazador de rayos distribuido, pero de manera simplificada, por lo que se puede considerar que estos trazadores tienen varias capas.

Estos algoritmos integran sobre toda la iluminancia que llega a cada punto de la superficie de un objeto y a tal resultado se le aplica una función de reflectancia (BRDF), que determina cuanta de esta iluminación llega a la cámara, y que por tanto, forma parte de la imagen sintetizada.

Este tipo de trazado de rayos aporta algunas ventajas sobre la simulación mediante trazadores de rayos tradicionales, consiguiendo que la iluminación global sea más fiel a la realidad, simulando efectos como las sombras suaves o la iluminación indirecta, entre otros, de manera natural, al contrario de lo que ocurre con los trazadores de rayos tradicionales, donde estos efectos tienen que ser añadidos específicamente.

Si se cuenta con modelos exactos y precisos de las fuentes de luz, óptica de la cámara, de las superficies y de las propiedades de los materiales, este tipo de simuladores pueden sintetizar imágenes prácticamente indistinguibles de las fotografías, y de hecho, se usan para generar imágenes de referencia con las que comparar las generadas por otros algoritmos de renderizado.

2.1.3.1 Primera capa: trazado de rayos simple

La finalidad de un software trazador de rayos es simular la interacción de la luz con los materiales de la escena que se trata de sintetizar, dando como resultado una imagen.

Si se descompone el plano donde se va a sintetizar la imagen en puntos (o píxeles), se puede encontrar una correspondencia de coordenadas entre el mundo en tres dimensiones de la escena y el plano en dos dimensiones de la imagen interpretada por nuestro cerebro, con lo que se pueden trazar líneas rectas entre esas correspondencias, estas líneas constituyen el camino de lo que llamaremos rayos.

En la naturaleza, las cámaras, o nuestros ojos, se ven afectadas por la radiación lumínica que proviene del exterior, directamente de las fuentes de luz, o reflejada por los objetos del entorno, esta radiación es la que se modela simplificándola en forma de rayos.

Sin embargo, sintetizar imágenes siguiendo cada rayo desde la fuente carece de sentido debido a que gran parte de los rayos calculados no incidirían en la cámara, y por tanto no formarían parte de la imagen. En un algoritmo típico de trazado de rayos, estos se "disparan" a través del plano de la imagen, en sentido contrario a como lo harían en la naturaleza, por lo que estos métodos se dice que están basados en el trazado de rayos inverso.

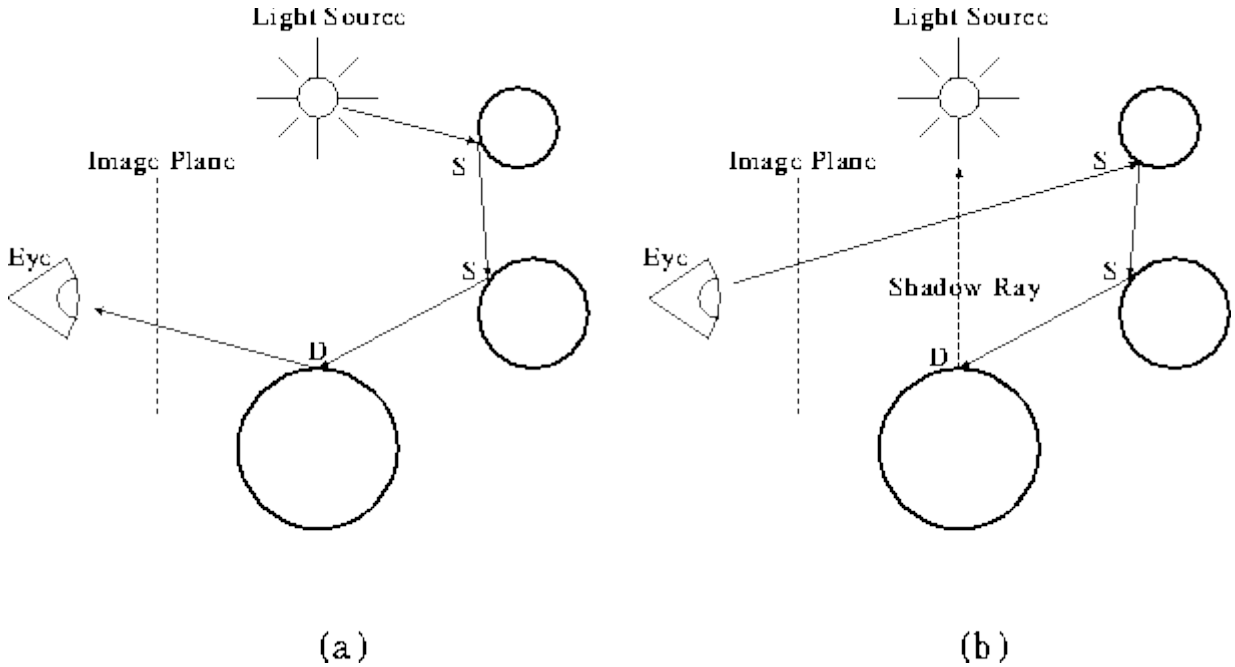


Ilustración 3: (a) Modelo físico, photon-tracing. (b) Trazado de rayos inverso simple.

Para cada rayo que se dispara, se calculan las incidencias de estos con los objetos, y se determina cuanta luz, proveniente del resto de la escena, incide en el punto.

2.1.3.2 Segunda capa: cálculo de la iluminación indirecta

Una vez calculada la iluminación directa, mediante una función probabilística basada en el método de Monte Carlo, se calcula aleatoriamente hacia que dirección disparar el rayo reflejado desde el punto de incidencia con las superficies de los objetos de la escena, y se determina si estos llegan al vacío, a otra superficie (que a su vez reflejaría un nuevo rayo reflejado, hasta un nivel arbitrario de profundidad), o a una fuente de luz.

El método de Monte Carlo es un método estadístico numérico, que se usa para aproximar expresiones matemáticas complejas muy costosas de evaluar con exactitud. Consiste en resolver un problema mediante procesos aleatorios, cuyo comportamiento simula un

fenómeno real gobernado por una distribución de probabilidad o para realizar cálculos costosos, como por ejemplo, en el caso que nos interesa aplicado a este proyecto, evaluar una integral. El uso de estos métodos simplifica el cálculo de las trayectorias de los rayos de luz, consiguiendo que las muestras seleccionadas de manera aleatoria por este método tengan más relevancia en el resultado del cálculo final. La función probabilística usada se determina por las propiedades ópticas de los materiales y las superficies de los objetos de la escena.

Este proceso se repite recursivamente hasta un nivel arbitrario de profundidad, y la iluminancia resultante del píxel por el que se ha disparado el rayo será la suma de las iluminancias calculadas en los rebotes, multiplicadas en cada nivel por un factor menor que la unidad, con lo que se consigue calcular la pérdida de energía en cada rebote.

2.1.3.3 Complejidad

Debido a la naturaleza discreta de las computadoras, se hace imposible de abordar el problema de evaluar completamente una integral de manera exacta, por lo que se debe de discretizar el espacio.

En el caso de la radiación lumínica, esto se consigue tomando muestras en forma de rayos, y es esta la primera de las simplificaciones que se van a llevar a cabo en este tipo de métodos para el cálculo de la iluminación. Evidentemente, cuanto más cerca se esté de calcular estas muestras de manera infinitesimal, más cerca se estará del resultado exacto, y por tanto de la evaluación completa de la integral.

Entonces, como una primera base para conocer la magnitud del problema, se tiene el tamaño o resolución de la imagen a generar, ya que, como se ha explicado en puntos anteriores, estos trazadores de rayos disparan los rayos a través de los píxeles de la imagen. Con la profundidad o nivel se definirá cual es el número de rebotes que se van a calcular para cada rayo, este nivel es arbitrario, de la misma manera que el tamaño de la imagen, pero también se debe de tener en cuenta al estudiar la cantidad de operaciones necesarias para el cálculo de la imagen.

Cada vez que uno de los rayos incide en una superficie, habría que conocer la suma infinita de toda la iluminación, teniendo como dominio de integración todo el hemisferio exterior (recordemos que el punto está situado en una superficie) alrededor del punto, y por las mismas razones descritas anteriormente, esto resulta computacionalmente inabordable. La solución adoptada aquí por los trazadores de rayos distribuidos es, de igual manera que a la hora de simplificar la radiación, muestrear de manera discreta sobre el hemisferio, lo cual, aunque simplifica el cálculo, no lo hace de manera eficiente, ya que al distribuir las muestras sobre el dominio integrador de manera regular no necesariamente todas esas muestras tendrán el mismo peso ni relevancia en el resultado del cálculo.

En cambio, los algoritmos de path-tracing encaran este problema haciendo uso de una función probabilística para calcular hacia dónde trazar un único rayo rebotado, de manera que sea más probable que la iluminancia que llega de esa dirección sea la más relevante en el cálculo final, haciendo que el número necesario de rayos a trazar para obtener el mismo resultado se vea disminuido considerablemente.

En ambos casos, a mayor número de muestras, mayor precisión del cálculo, pero si se comparan, a mismo número de muestras en ambos métodos, estadísticamente es más probable que el de path-tracing sea más preciso, ya que estas muestras han sido estadísticamente mejor elegidas.

Debido a esta manera probabilística de calcular los rayos reflejados, cada vez que se lanza un mismo rayo el resultado no es exactamente el mismo, por lo que repetir el cálculo y realizar el cálculo de la media de resultados, nos dará un resultado más preciso, con lo que este número de repeticiones también será una magnitud a tener en cuenta para hacerse a la idea del tamaño del cálculo.

Todavía, después de aplicar estas simplificaciones, la cantidad de operaciones a realizar es enorme, y resulta lógico estudiar formas de tratar ya directamente este cálculo, ya que estos cálculos aunque abordables, siguen teniendo un coste en tiempo elevado. Sin embargo, las computadoras actuales reúnen una serie de características que permiten recurrir a ciertas estrategias para obtener el resultado del cálculo en una cantidad de tiempo menor, entre ellas, la capacidad de realizar varios cálculos de manera concurrente.

2.2 Computación paralela

2.2.1 CPUs

La unidad central de procesamiento o CPU (del inglés: *Central Processing Unit*), hace referencia al elemento de un computador que interpreta las instrucciones y procesa los datos de los programas.

Esencialmente, están constituidas por: registros, unidad de control, unidad aritmético-lógica, y posiblemente una unidad de cálculo en coma flotante (conocida como coprocesador). Aunque las arquitecturas de las CPUs actuales tienden a ser mucho más complejas, ya que han sufrido una de las evoluciones más impresionantes y rápidas en la historia de la tecnología.

Los microprocesadores actuales poseen uno o varios niveles de la conocida como memoria caché, que se usa como almacén intermedio entre las CPUs y la memoria principal del sistema, y es mucho más rápida que esta, aunque más pequeña, de hecho, a mayor nivel: mayor tamaño, menor velocidad, y mayor tiempo de acceso. Por ello, se suele usar para que la CPU tenga al alcance directamente ciertos datos que probablemente vaya a usar con

más asiduidad o de forma más inmediata, priorizando por niveles.

Las subunidades que componen los procesadores realizan cálculos a nivel de palabra, es decir, un conjunto de bits a los que se les aplica la misma operación. Entonces el nivel más bajo de paralelización simplemente lo define el ancho de palabra, o número de bits que la componen. Actualmente el tamaño de palabra más común se trata del de 64 bits, aunque históricamente se han usado palabras de 32, 16, 8 y las primeras unidades de proceso lo hacían con palabras de 4 bits.

Aprovechando la independencia existente entre las distintas subunidades que componen los microprocesadores, los hilos de ejecución se diseñan conforme al concepto de segmentación. Esta técnica consiste en que las instrucciones se descomponen en varias etapas, en cada una de las cuales cada instrucción hace un uso exclusivo de una subunidad dejando libres las restantes, pudiendo ser utilizadas por otras etapas de otras instrucciones. Esto supone otro nivel de paralelización, pues se puede ver que, aunque es cierto que no acaban a la vez, hay varias instrucciones ejecutándose de manera concurrente. Este tipo de paralelismo viene dado por la propia arquitectura del procesador, y los compiladores recientes tienen la capacidad de aplicar ciertas técnicas para optimizar el código máquina que generan, aprovechando las posibilidades de la arquitectura para la que están generando ese código. A este nivel se puede encontrar todo un mundo, perteneciente al ámbito de la arquitectura y diseño de computadores, que mezcla a la vez hardware y software de bajo nivel, con mucho por explorar y de gran interés, pero tan extenso que se escapa de los límites de este estudio.

Los microprocesadores de las computadoras modernas integran varios núcleos, por lo que en realidad se trata de multiprocesadores, y por tanto, ya presentan cierta capacidad de realizar cálculos en paralelo, compartiendo la memoria principal del sistema, por lo tanto presentan un paralelismo de memoria compartida, si bien habría que tener en cuenta la coherencia de caches entre unidades.

Una solo núcleo también puede tener la capacidad de ejecutar múltiples hilos de ejecución, esta característica, conocida como multithreading simultáneo, o mediante la denominación comercial de Intel, Hyperthreading, no se trata de varios hilos de cálculo completamente independientes, ya que, aunque virtualmente se tengan varios procesadores, en este caso la totalidad de la unidad de proceso no se encuentra físicamente replicada de forma completa sino solo partes específicas de esta.

2.2.2 GPUs

La unidad de proceso gráfico o GPU (Graphics Processor Unit) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, en principio usado para aligerar la carga de trabajo del procesador central de un sistema en aplicaciones como

videojuegos. Al liberar de esta carga a la CPU, la capacidad de esta puede dedicarse a otro tipo de cálculos simultáneos de utilidad para esas aplicaciones, como cálculos de la mecánica o la inteligencia artificial en el mundo del videojuego.

Las GPUs actuales son muy potentes, y pueden llegar a frecuencias de reloj comparables a las alcanzadas por las CPUs, aunque no es reemplazable una por otra, debido a la alta especialización de las GPUs. De hecho, esta especialización es la razón de su alta potencia, ya que al estar pensadas para realizar una tarea en concreto, es posible dedicar más espacio físico, en el silicio, para componentes con los que llevar a cabo esa tarea de manera más eficiente.

Las aplicaciones gráficas conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo completamente independientes, por lo que las GPUs suelen consistir en cientos de procesadores shader unificados, que son capaces de actuar como vertex shaders, pixel shaders, o fragment shaders. Adicionalmente, en las GPUs se encuentra una cantidad de memoria RAM usada por las propias unidades de cálculo.

En el pipeline gráfico, los vertex shaders tendrían el papel de calcular las operaciones que se deben aplicar a los vértices de los triángulos que componen los objetos, y una vez aplicado esto y hecha la traslación a píxeles, los pixel shaders se encargan de calcular texturas, iluminación, etc., también es en esta etapa cuando se aplican efectos como el antialiasing o efectos ópticos propios de la lente de la cámara sintética. Una vez realizado, se almacena temporalmente en la caché, desde la cual otras unidades, llamadas ROP, preparan los píxeles para su visualización.

Debido a las diferentes arquitecturas de GPUs, en un inicio estas se programaban en un lenguaje ensamblador específico para cada arquitectura, más tarde la tecnología evolucionó creando APIs específicas para gráficos, de las cuales cabe destacar el estándar abierto OpenGL, o la biblioteca propietaria DirectX.

2.2.2.1 GPGPU

Con el objetivo de aprovechar la potencia de cálculo de las GPUs para realizar cálculos fuera del ámbito de los gráficos, nace el concepto de la computación de propósito general en unidades de proceso gráfico, o GPGPU (del inglés: General-Purpose Computing on Graphics Processing Units).

Las características especiales de cálculo de las GPUs, tales como la especialización en coma flotante, o la manera masivamente paralela de trabajar, las hacen idóneas, además que para su utilización en la generación de gráficos, para los cálculos científicos en diversos campos, como por ejemplo la simulación.

Precisamente, debido a estas características, los algoritmos se deben de programar

pensando en su ejecución en estos sistemas, y para ello se han creado ciertos lenguajes o extensiones de lenguajes ya existentes, como por ejemplo y la más extendida, *CUDA*, que es una extensión para el lenguaje C, diseñada para codificar programas con cálculos de propósito general para las GPUs de nVidia. AMD (antes ATI), creó su propia versión de esta tecnología y la denominó *Close to Metal*.

También en este aspecto habría que nombrar el lenguaje e interfaz, OpenCL, que persigue la creación de una tecnología con la que ser capaz de codificar programas paralelos de manera independiente a la arquitectura de las máquinas donde se ejecuten, pudiendo mezclar CPUs y GPUs de diversas arquitecturas, independientemente de que se traten de un fabricante u otro.

2.2.3 Clústeres

Las unidades de cálculo, que se han descrito en puntos anteriores, unidas a una memoria principal, unidades de almacenamiento, y capacidad de entrada/salida consisten una computadora. A su vez estas computadoras, dotadas con capacidad de comunicación en red, pueden agruparse para formar un grupo de computadoras interconectadas, que programadas y coordinadas de la manera oportuna forman un cluster. El papel de cada computadora en una formación de este tipo recibe el nombre de nodo.

La agrupación de este modo, permite crear sistemas con características que no serían que una computadora por separado no sería capaz de proveer, habitualmente, estas características son las de: alto rendimiento, alta disponibilidad y alta eficiencia.

Mientras que las características relativas a la eficiencia y rendimiento del sistema hacen referencia a las capacidades que tiene para realizar cálculos en el menor tiempo posible, la disponibilidad denomina a la capacidad de hacer frente a fallos, ya que por ejemplo, el fallo de uno de estos nodos no comprometería el funcionamiento del sistema y este seguiría operativo apoyándose en el funcionamiento del resto de nodos.

Adicionalmente a los nodos, un cluster de computadoras no es nada sin una buena red de interconexión, un almacenamiento, y un middleware.

Con respecto a la red de interconexión, ésta directamente puede tratarse desde una red ethernet básica, hasta soluciones más avanzadas tecnológicamente, como *infiniband* o *mirinet*. Evidentemente, cuanto más ancho de banda y menos latencia tenga la red, menos impacto negativo tendrá en el rendimiento del sistema.

El almacenamiento es un tema crítico si la finalidad del cluster es el análisis o la generación de gran cantidad de datos. Existen sistemas de ficheros específicos para su uso en clústeres, como *GlusterFS*, o *LUSTRE*, hasta soluciones más avanzadas, pero que ya suponen un cluster en sí únicamente para el servicio de almacenamiento, como las basadas

en *Ceph*. Los sistemas de ficheros desplegados en un cluster suelen tratarse de sistemas de ficheros distribuidos, y algunos de los middleware o frameworks para la computación masiva se aprovechan de esta característica analizando la localidad de los datos.

El middleware es una pieza de software que proporciona la capacidad de ver a un cluster como una máquina completa, en vez de como varias máquinas independientes, aunque la realidad física sea esa. Este software se encarga de balancear la carga entre nodos, migración de procesos, gestión de colas de trabajo, y la asignación de prioridades a procesos, entre otras acciones.

Se puede hacer una distinción en cuanto a estos sistemas, ya que inicialmente se buscaba realmente dar la imagen de un sistema operativo único, con un kernel distribuido, ejemplos de esto podrían ser *OpenSSI*, u *OpenMosix*.

Sin embargo, la tendencia actual es dotar a los nodos de capacidad de cálculo en paralelo, aun sin perder su sistema operativo local, soluciones de este tipo son las que proporcionan, por ejemplo, los proyectos *Apache Hadoop* o *Condor*. También es posible que una forma básica de middleware esté en el propio algoritmo de cálculo, y no tener que recurrir a soluciones a nivel de sistema, como por ejemplo haciendo uso de técnicas de paso de mensajes.

2.2.4 Paralelización

La idea básica detrás del concepto de paralelización de algoritmos es la realización de varios cálculos de manera concurrente. No obstante, hay que tener en cuenta varios aspectos y consideraciones a la hora de paralelizar los cálculos, ya que aunque la idea es sencilla, en la práctica se encuentran problemas debido a la propia arquitectura de memoria de los sistemas capaces de realizar estos cálculos en paralelo, las dependencias de los resultados, la localidad de los datos, o la propia capacidad de paralelización del algoritmo, entre otros, que hacen que la paralelización en sí misma sea otro campo más a estudiar e investigar.

Partimos de la base de que un hilo de ejecución hardware, únicamente se aprovecha en su totalidad si está realizando cálculo activamente y no esperando, el tiempo desaprovechado decimos que es tiempo ocioso. Esta afirmación de base que resulta tan trivial en el caso de una ejecución secuencial, no es tan evidente en el ámbito de la computación paralela, aunque un ejemplo nos permitirá visualizar este hecho: si tenemos dos unidades de cálculo, estaremos aprovechando totalmente la capacidad de cálculo cuando ambas unidades estén realizando cálculos activamente, ya que si una de ellas sólo estuviera realizando cálculos la mitad del tiempo, la carga de trabajo recaería únicamente sobre la otra unidad, desperdiciando “un cuarto” de la capacidad total de cálculo, por supuesto este ejemplo es extrapolable a N unidades de cómputo.

El uso del entrecomillado en el ejemplo anterior no es casual, ya que la teoría nos marca unos límites de rendimiento. Parece claro que lo ideal sería que la aceleración conseguida por la paralelización fuese la lineal, es decir, que duplicar el número de unidades de procesamiento reduzca el tiempo de procesado a la mitad y duplicarlo una segunda vez lo reduciría a la cuarta parte. Sin embargo muy pocos algoritmos logran esa aceleración ideal, la mayoría aceleran de manera lineal para un número pequeño de unidades de proceso y pasa a ser constante para un número más elevado de unidades.

Este fenómeno queda plasmado en la teoría por la *ley de Amdahl*, que establece:

> *"La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente."* [Amd67]

Y se aplica particularmente al caso de la paralelización debido a que existen secciones de los algoritmos que son paralelizables y otras que no lo son, las cuales necesariamente han de ser secuenciales, entonces, aunque se consiguiera reducir el tiempo de las secciones paralelizables de manera infinita (tiempo de ejecución nulo), el límite lo marcaría el tiempo necesario para calcular las secciones secuenciales.

Formalmente, si α es la fracción de tiempo que un algoritmo invierte en secciones no paralelizables, y P el número de unidades de proceso, tenemos que la máxima aceleración alcanzable viene dada por:

$$S = \frac{1}{\alpha} \qquad = \lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha}$$

Es interesante recalcar el hecho de que la paralelización se puede presentar en varios niveles:

- Paralelización a nivel de datos: aplicar el mismo algoritmo a fragmentos del total de datos, actuando cada unidad de proceso sobre un fragmento diferente.
- Paralelización a nivel de tareas: aplicar distintos algoritmos al total de los datos, donde cada unidad de proceso realiza un cálculo diferente.
- Paralelización a nivel de instrucción: haciendo uso de la segmentación de instrucciones apoyada en la independencia que poseen las distintas subunidades que componen las unidades de proceso.
- Paralelización a nivel de bit: ampliando el tamaño de palabra con el que trabaja una unidad de proceso.

El paralelismo conseguido tanto a nivel de instrucción como a nivel de bit vienen

determinados por la arquitectura y el diseño de cada unidad de proceso, y por tanto cambian con la máquina, en cambio, el paralelismo a los otros dos niveles lo define el propio algoritmo a afrontar. La forma de tratar esos tipos de paralelismo es diferente y hay que estudiar que características posee el problema a solucionar de manera específica, ya que resulta imposible establecer soluciones globales que funcionen universalmente para todos los algoritmos. Aunque se describirá en una sección dedicada, se puede precisar que el algoritmo usado para llevar a cabo este estudio, posee un paralelismo a nivel de datos.

Existe una distinción clara entre las organizaciones de memoria usadas por los sistemas con capacidad de procesamiento en paralelo: en un lado el uso de un espacio de direccionamiento común, y en el otro, uno privado a cada unidad de proceso. Esta distinción fuerza también a tener en cuenta la capacidad de comunicación entre estas, ya sea para obtener los datos de entrada para los cálculos, o bien para el almacenamiento de los resultados.

2.2.4.1 Memoria compartida

En los sistemas de memoria compartida, cada unidad de proceso tiene acceso a toda la memoria, ya que cuentan con un espacio de direccionamiento compartido, común a todas las unidades.

Físicamente, esta memoria puede estar centralizada o repartida (distribuida) entre las unidades de proceso.

Las grandes ventajas que aporta esta arquitectura son la simplicidad de las implementaciones de los algoritmos que hacen uso de ellas, y que no se hace necesario especificar en el código la comunicación para el traspaso de datos entre unidades de proceso, de hecho, en el caso de arquitecturas con acceso a una única memoria física, esa comunicación es inexistente. En cambio tiene una serie de inconvenientes, como podría ser la coherencia de caches, en el caso de la memoria compartida-distribuida, y sobre todo en la limitada escalabilidad de ambos sistemas.

2.2.4.2 Memoria distribuida

Las unidades de proceso que forman parte de los sistemas basados en memoria distribuida tienen su propia memoria local.

Para que una unidad de proceso pueda operar con datos que otra unidad tiene en su memoria local, es necesaria una transmisión de esta información de una a otra, ya que no tienen acceso a la misma memoria, y este aspecto es uno de los inconvenientes de esta arquitectura. En cambio, este sistema ofrece una escalabilidad mucho mayor, que no sería posible desde el enfoque de la arquitectura de memoria compartida.

2.2.4.3 Comunicación

Necesariamente, las unidades de proceso que llevan a cabo un cálculo en paralelo se tienen que comunicar de algún modo, pues necesitan o bien traspasar datos de una unidad a otra, si estas no comparten un direccionamiento común, o en el caso de las arquitecturas de memoria compartida para indicar a otra unidad de proceso la posición de memoria del resultado calculado por una unidad; o bien por el simple hecho de coordinarse: indicar la finalización de un cálculo por parte de una unidad cuyo resultado es requerido como entrada para los cálculos de otra, para señalar y coordinar el inicio de los cálculos, o para otros aspectos.

Según la naturaleza del algoritmo a paralelizar, y a las estrategias para repartir el trabajo entre las unidades de proceso, esta comunicación será más o menos intensiva, por lo que el medio (hardware) por el que se van a transmitir estas comunicaciones también tendrá un impacto en la eficiencia de estos algoritmos.

2.2.4.4 Tecnologías de paralelización

OpenMP

OpenMP es una interfaz para la programación de aplicaciones multiproceso en plataformas de memoria compartida y permite añadir concurrencia a los programas escritos en C, C++ y Fortran con un modelo de ejecución fork-join, este modelo de ejecución consiste en bifurcar un proceso en varios hilos y después agregar el resultado de todos ellos mediante una función que los integra.

MPI

MPI (del inglés: Message Passing Interface), es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para usarse en programas que explotan las características de los sistemas capaces de realizar cálculos en de forma concurrente. El paso de mensajes es una técnica que aporta sincronización entre procesos y que permite la exclusión mutua, de manera similar a la utilización de semáforos o monitores. La implementación más relevante de este estándar es OpenMPI.

Hadoop

Apache Hadoop es un framework creado en el lenguaje Java para soportar aplicaciones distribuidas y que permite escalar al nivel de miles de nodos y a petabytes de datos, por lo que se usa para analizar cantidades enormes de datos, o lo que se denomina Big Data.

La arquitectura de este sistema consiste en dos partes diferenciadas, en un lado se tiene un sistema de ficheros, HDFS, que hace el trabajo de replicar y localizar los bloques de datos para hacerlos accesibles de forma concurrente a la otra parte, la encargada del cálculo, MapReduce.

2.3 Path tracer sobre un cluster heterogéneo de CPUs y GPUs

Como último punto de este capítulo, y habiendo presentado las distintas tecnologías y técnicas anteriores, se puede pensar en unirlas para un fin común.

Así pues, para este proyecto de fin de carrera, se plantea el estudio del rendimiento al generar imágenes sintéticas simulando la interacción de la luz con los materiales mediante un software de trazado de rayos basado en path tracing, con capacidad de realizar cálculos en paralelo y desplegado sobre un cluster de varias computadoras, disponiendo, cada una de ellas, tanto de CPUs con varios núcleos, como GPUs.

3 Descripción del sistema

3.1 Hardware

El sistema en el que se ha desplegado el proyecto consiste en un cluster de computadoras conectadas en red, el software de trazado de rayos y otros software que permiten la comunicación coordinación y tráfico de datos entre ellas. Este cluster se compone de máquinas propias del Grupo de Informática Gráfica Avanzada y de equipos de los laboratorios docentes del departamento de informática e ingeniería de sistemas.

Estas computadoras tienen distintas características y unidades de proceso, en las cuales se incluyen a las CPUs y/o las GPUs, que a su vez pueden pertenecer a distintas arquitecturas, debido a esto, se considera que el sistema de trazado de rayos desplegado en este entorno es heterogéneo.

3.1.1 Computadoras

En esta sección se presentan las computadoras que han formado parte del sistema desplegado para la realización de este estudio junto a sus características principales.

Nombre	CPU				GPU				S.O.
	Modelo	Núcleos	Hilos	Memoria	Modelo	Núcleos	Hilos	Memoria	
Equipos del laboratorio 0.01	Core i5-3470	4	4	4 GB	N/A	N/A	N/A	N/A	CentOS 6.7
Annwn	Core i7-4790	4	8	8 GB	N/A	N/A	N/A	N/A	Mageia 6
Beatrix	Core i7-4770	4	8	8 GB	GeForce GTX 660	5	960	2GB	Mageia 6
Amelia	Core i7-2600	4	8	6 GB	GeForce GTX 570	15	480	1280MB	Mageia 6
Grendel	Core2 Quad Q9450	4	4	4 GB	Tesla C1060	30	240	4GB	CentOS 7.2
					Tesla C1060	30	240	4GB	
					Tesla C1060	30	240	4GB	
					Tesla C1060	30	240	4GB	
Wargo	Core2 Quad Q9450	4	4	4 GB	Tesla C1060	30	240	4GB	CentOS 7.2
					Tesla C1060	30	240	4GB	
Lugh	Pentium 4	1	2	3 GB	N/A	N/A	N/A	N/A	Mageia 6
Bran	Pentium 4	1	2	3 GB	N/A	N/A	N/A	N/A	Mageia 6

Tabla 1: Características de las computadoras del sistema

Estas computadoras se encuentran interconectadas mediante la red ethernet institucional

de la Universidad de Zaragoza.

3.2 Software

Además de las máquinas, se utiliza el propio software de trazado de rayos, el cual se detalla más adelante, y que se ha modificado para repartir la carga de trabajo paralelizando los cálculos, para este fin se ha utilizado la biblioteca OpenMPI, que es una implementación de código abierto de MPI (*Message Passing Interface*) [MPI]. El motor trazador de rayos y la adaptación al cálculo en paralelo usados en este proyecto están escritos en el lenguaje de programación C++11, aunque en la adaptación del código para realizar la paralelización se han utilizado los bindings de MPI para C, ya que los de C++ se consideran obsoletos por el MPI Forum, la organización que redacta el estándar MPI, al no aportar ventajas sobre los de C y a que el compilador de C++ es totalmente compatible con estos.

Los sistemas operativos que gestionan las máquinas del sistema están basados en Linux, aunque no se trata en todos los casos de la misma distribución, ya que se han usado los sistemas con los que ya contaban, encontrándose entre ellos: Debian, Ubuntu, Mageia y CentOS. Esto permite que las rutas de acceso tengan la misma estructura, que se puedan conectar entre ellos mediante las tecnologías NFS (*Network File System*) para la compartición del directorio de trabajo, SSH para el acceso a una shell remota, y que tengan disponibles las herramientas y bibliotecas de OpenMPI.

3.2.1 C++ 11

C++ 11 es la revisión del lenguaje de programación C++ aprobada en agosto de 2011 sobre la revisión anterior, C++03, y añade ciertas evoluciones con respecto a este, entre otras y la que más ha convenido en la concepción de este motor de renderizado es la inclusión del modelo de hilos para la ejecución concurrente de código en la biblioteca estándar.

C++ es un conocido lenguaje de programación que ofrece los paradigmas de programación orientada a objetos y programación estructurada, que surge como evolución del lenguaje C.

3.2.2 OpenMPI

Esta biblioteca se ha usado para escribir el código de las tareas de sincronización y comunicación entre procesos del sistema, es decir, para el desarrollo de las funciones esenciales que han permitido la paralelización del código en varias máquinas.

Se trata de una implementación libre y de código abierto del estándar de paso de mensajes MPI. Este estándar define la sintaxis y la semántica de las funciones que componen la biblioteca, entre las que se hallan las propias para envío y/o recepción de mensajes y datos, identificación y sincronización de los procesos. También se definen los tipos de datos y estructuras diseñadas para su correcto funcionamiento.

Las características principales de esta tecnología son la estandarización, la portabilidad (posibilidad de implementaciones en distintas arquitecturas, lenguajes y sistemas), el rendimiento, la amplia funcionalidad y la existencia de implementaciones libres.

Esta biblioteca es usada por la mayor parte de los supercomputadores pertenecientes a la lista de los 500 más rápidos del mundo, la TOP500, en su mayoría concebidos para ayudar en el cálculo de aplicaciones científicas de alto rendimiento enfocadas a la investigación y estudio por universidades y otras instituciones.

3.3 Recursos

El acceso a los datos necesarios para generar la escena, tales como: modelos, texturas, mallas de puntos, etc... se realiza a través un directorio de trabajo compartido por todas las máquinas del sistema mediante NFS, haciendo que todos los nodos tengan un espacio de almacenamiento común, aunque este únicamente se usa como fuente de recursos y almacenamiento final del resultado, nunca como almacén de intercambio de datos de cálculo entre los nodos, esto, aunque probablemente más sencillo de gestionar, tendría un impacto negativo en el rendimiento del sistema, debido a ello, estas transmisiones de datos se llevan a cabo mediante las funciones propias de MPI para esta finalidad.

Con el objetivo de simplificar la gestión de las rutas de acceso a estos recursos se ha optado por mantener el mismo punto de montaje en todas las máquinas.

3.4 Control y monitorización

Las mediciones de tiempos y el seguimiento de la ejecución de los programas se ha realizado usando las herramientas típicas de los entornos Linux, tales como top, htop, time; o bien mediante la instrumentación del propio software desarrollado.

Se han escrito funciones dedicadas a extraer tiempos y calcular datos estadísticos para facilitar la tarea de la realización de pruebas y experimentos, y la extracción de conclusiones al respecto.

4 Diseño y paralelización del trazador de rayos

4.1 Trazador de rayos ALEPH

4.1.1 Descripción

El software ALEPH [Mag03] es un sistema de simulación física de la iluminación y generación de imágenes sintéticas realistas desarrollado en el GIGA como herramienta de investigación.

El sistema utiliza modelos físicos en todas las etapas de la simulación, por lo que los resultados se pueden utilizar de forma predictiva y fiable, no solo a nivel de imagen, sino a nivel de resultados numéricos de la iluminación en un entorno. El algoritmo utilizado está basado en el trazado de rayos distribuido, utilizando BRDFs físicas y con especial incidencia en la corrección no solo visual, sino física, de los resultados.

En la actualidad el sistema está en fase de rediseño e implementación de ciertos algoritmos, originando un nuevo sistema denominado ALEPH/FTL (Faster Than Light).

Los motivos para ese rediseño son entre otros:

- La mejora del rendimiento general, aumentando la velocidad de render.
- La utilización del algoritmo de path-tracing, en lugar de trazado de rayos distribuido.
- La implementación del uso de GPUs para realizar la simulación en paralelo con la CPU del sistema.
- La conversión de partes del sistema a la nueva funcionalidad estándar aparecida en C++11 (gestión de multiproceso simétrico, hilos, mutexes, temporizadores...) que permiten que sea más portable y la simplificación del código.

Hasta el momento se ha dedicado el esfuerzo de reimplementación a la mejora de los tiempos de intersección y trazado de rayos, y la implementación de los modelos de iluminación, dejando pendientes temas como el texturado, efectos volumétricos, etc, que mejorarían el aspecto visual de la imagen.

4.1.2 Algoritmo simplificado

```
Color TracePath(Ray r, depth) {
  if (depth == MaxDepth) {
    return Black;  // Suficientes rebotes.
  }

  r.FindNearestObject();
  if (r.hitSomething == false) {
```

```
    return Black;  // No hubo intersección.
  }

  Material m = r.thingHit->material;
  Color emittance = m.emittance;

  // Escoger dirección aleatoria y continuar.
  Ray newRay;
  newRay.origin = r.pointWhereObjWasHit;
  newRay.direction = RandomUnitVectorInHemisphereOf(r.normalWhereObjWasHit);

  // Calcular la BRDF para este rayo (asumir reflección difusa)
  float cos_theta = DotProduct(newRay.direction, r.normalWhereObjWasHit);
  Color BDRF = 2 * m.reflectance * cos_theta;
  Color reflected = TracePath(newRay, depth + 1);

  // Aplicar la ecuación de renderizado aqui.
  return emittance + (BDRF * reflected);
}
```

4.2 Paralelización

Como se ha tratado en apartados anteriores, la idea detrás del aprovechamiento de las capacidades computacionales del cluster reside en la posibilidad de realizar cálculos de forma concurrente.

Para que la capacidad de cálculo se aproveche de manera eficiente, el objetivo a conseguir es repartir el trabajo de tal manera que las unidades de proceso estén realizando activamente trabajo de cálculo durante todo el tiempo que dure el cálculo completo, evitando en lo posible los tiempos ociosos de alguna de ellas.

Analizando el problema de síntesis de imagen y su algoritmo, vemos que presenta un paralelismo de datos, con lo que la forma de repartir el trabajo será dividiendo directamente la imagen a calcular, por simplicidad, esta división se ha realizado en franjas horizontales.

El trabajo resulta tan sencillo de dividir debido a la independencia de los resultados de los cálculos, ya que cada muestra (rayo) no necesita ningún otro resultado de otro cálculo para ser procesada, y ya que los rayos se disparan una o varias veces para cada píxel de la imagen, podemos asignar píxeles a trabajos, y por simplicidad, estos píxeles han sido seleccionados por franjas horizontales que suponen un área, que consiste un fragmento de imagen.

El sistema hardware sobre el que se ha realizado el estudio, posee la capacidad de realizar cálculos en paralelo a varios niveles: a nivel de unidad de proceso, de nodo, o de cluster; así que en los siguientes subcapítulos se explicarán las distintas técnicas empleadas.

4.2.1 Nivel de unidad de proceso

El motor de trazado de rayos que se ha usado como base para este proyecto presentaba ya una paralelización a nivel de hilos, que originalmente se lanzaban a razón de uno por núcleo de la CPU, repartiendo la imagen de manera equitativa dividiéndola en tantas franjas iguales como hilos de ejecución soporta nativamente el procesador. Esto resulta relativamente sencillo debido al hecho de que las subunidades de una misma unidad de proceso, los núcleos, suelen ser prácticamente idénticas y por tanto tienen un rendimiento similar.

No obstante, en la evolución de este estudio y al comprobar las ventajas de un reparto de trabajo más dinámico, que se detallará en el apartado dedicado a la paralelización a nivel de cluster y llamaremos *cola de trabajos*, se ha terminado por adaptar este tipo de reparto también entre los núcleos de una unidad de proceso, con el pretexto de conseguir un reparto de trabajo mejor balanceado (más equilibrado).

4.2.2 Nivel de nodo

A nivel de una máquina completa, se puede disponer de varias unidades capaces de realizar cálculos, por lo que el sistema también debe de repartir el trabajo entre estas unidades.

El objetivo en este reparto de trabajo sería asignar a cada unidad de proceso una cantidad de trabajo proporcional a la capacidad relativa con respecto a la capacidad total de unidades que tiene el nodo.

La estimación de esa capacidad relativa supone un problema en sí misma, y se han desarrollado (y descartado) varios procedimientos enfocados a tal fin. Esta estimación se podría realizar conociendo los datos de las propias unidades de proceso, como la frecuencia, el número de núcleos y/o hilos de procesamiento; o de una manera secundaria haciendo un pequeño benchmark. Todas estas técnicas tienen sus ventajas e inconvenientes, pero ninguna de las desarrolladas ha dado unos resultados realmente satisfactorios.

En la implementación actual del software trazador la velocidad de cada CPU corresponde al número de hilos de proceso totales: suma de todos los hilos de todos los núcleos. En cambio para la estimación de la potencia de la GPU, se toma el propio número de núcleos de la GPU y un factor multiplicativo dependiente de la generación de la arquitectura de esta.

Aunque es una técnica sencilla y que obtiene unos resultados proporcionales, esta no llega a hacer un reparto eficiente del trabajo y provoca desbalanceos en la carga de trabajo que

penalizan el tiempo de cálculo, como se pone de manifiesto en las pruebas llevadas a cabo para la realización de este estudio.

4.2.3 Nivel de cluster

Interconectando varios nodos, tenemos un cluster, lo que supone otro nivel de paralelismo, donde también se deben de definir las estrategias para la gestión de esos cálculos concurrentes.

Tomando la paralelización de los niveles más bajos como referencia, teniendo en cuenta las necesidades de la paralelización a nivel de nodos a través de la red, y que al no tener una memoria compartida se hace necesaria una comunicación adicional, se ha construido una capa de control dentro de la biblioteca, por encima de las capas ya existentes, donde entre otros métodos, se definen los esenciales para repartir el trabajo y la puesta en común de resultados.

En esta fase del desarrollo es donde se han usado las funciones de la biblioteca MPI, junto con las ya usadas en el código de base relativas a la ejecución concurrente mediante hilos de ejecución de la biblioteca estándar de C++11.

4.2.3.1 Reparto de trabajo

Debido a la naturaleza del problema, y de los métodos empleados, nos encontramos que para paralelizar el trabajo simplemente se ha de dividir la imagen, pudiendo procesar con cualquier unidad cualquier segmento de esa imagen como si se tratase de una imagen completa, sin tener dependencias de resultados de unas a otras, además, debido a que el sistema de ficheros del que se extraen los datos relativos a los componentes de la escena es el mismo (montado en todas las máquinas por medio de NFS), únicamente se hace necesaria la comunicación entre las unidades para la puesta en común del resultado y para comunicar a cada unidad de proceso los límites en los que debe de trabajar, aunque esto último no es necesario en todos los casos.

Los datos necesarios para generar la escena, tales como modelos de objetos, mapas de entorno de iluminación y otros, son accesibles por todos los nodos, y cada uno de ellos genera la escena de forma local en su propia memoria, de manera que esta queda replicada en cada uno de los nodos que forman el cluster, aunque el cálculo de la imagen no se realice enteramente en cada uno de ellos, sino que cada nodo calcula el área de la imagen que le ha sido asignada mediante las distintas estrategias implementadas.

En la adaptación se ha considerado que el proceso que hace de maestro, al ejecutarse también en un nodo con capacidad de cálculo, además de ejercer los papeles de director de reparto de trabajo y de receptor de resultados provenientes de los demás nodos, también aporta resultados de cálculo como cualquier otro proceso integrante del sistema.

Por simplicidad, la división de la imagen se realiza en franjas horizontales y se han considerado tres estrategias para el reparto de trabajo entre las unidades de proceso: equitativa, proporcional a la capacidad y mediante una cola de trabajos.

Equitativo

La estrategia más sencilla de todas es la del reparto equitativo, la estrategia es simple y no es necesaria ninguna comunicación para el reparto de trabajo, ya que cada unidad de proceso dispone de los datos necesarios para poder calcular cual es su segmento a procesar con las siguientes formulas:

$$inicio = identificador\,del\,proceso \cdot \frac{altura\,total\,de\,la\,imagen}{numero\,de\,procesos}$$

$$final = inicio + \frac{altura\,total\,de\,la\,imagen}{numero\,de\,procesos} - 1$$

Proporcional a la capacidad

Otra estrategia que se pensó útil es la del reparto proporcional a la capacidad de cada máquina, pero para ello se necesita algo de comunicación, ya que los procesos que se trabajan en cada máquina deben enviar la capacidad de esta a uno de los procesos que actúa como encargado de calcular la capacidad relativa de cada máquina respecto a la total del sistema, con estos datos, el proceso maestro calcula el tamaño de las franjas a procesar y se envía el inicio y el final de cada una a los demás procesos, que se han quedado a la espera de recibir tales datos para poder comenzar a realizar el trabajo de cálculo, no obstante, la eficiencia de este método depende de la exactitud a la hora de medir o estimar la velocidad de las unidades de proceso.

Las funciones con las que se estima la capacidad total de cada nodo tienen como base las mismas que se usan para el reparto entre unidades de proceso a nivel de nodo, de hecho, la capacidad total de una máquina que se envía al proceso maestro es la suma de las velocidades de todas sus unidades de proceso. Por tanto, los problemas experimentados en cuanto a ese reparto son también patentes a este nivel de reparto.

Cola de trabajos

La estrategia más elaborada que se ha implementado es una cola de trabajo. Usando esta estrategia, la imagen se divide en un numero arbitrario de franjas y se arranca un hilo concurrente en uno de los procesos, que será el encargado de repartir el trabajo y hará las veces de maestro, este hilo escucha peticiones y responde a ellas con el inicio y final de la siguiente franja a calcular. Los hilos principales de todos los procesos se encargan de realizar esas peticiones, y conociendo los límites de la franja a calcular comienza con el trabajo de cálculo.

4.2.3.2 Puesta en común de resultados

La otra parte necesaria de la comunicación está en la puesta en común del resultado a través de mensajes entre procesos, pues se ha considerado que uno de los procesos es el que debe de tener la imagen calculada por completo en memoria, para después escribirla en disco o realizar las acciones que fueran pertinentes. Se puede pensar también en realizar esta puesta en común de resultados directamente escribiendo cada unidad de proceso sus resultados en el directorio de trabajo, ya que éste está compartido por todas las máquinas, pero esto implicaría un tratamiento posterior volviendo a leer cada uno de estos resultados para su utilización u obtención de la imagen completa. Por el poco interés que tiene esta opción debido a la pérdida de rendimiento al pasar los datos por el almacenamiento físico, se ha optado únicamente por la anterior.

En la solución adoptada, el proceso que tiene el papel de maestro arranca un hilo concurrente que queda a la escucha y recibe los resultados de los cálculos de los demás procesos, esto permite que el proceso maestro pueda seguir calculando su parte de trabajo en el hilo principal.

Cuando una unidad de proceso termina de realizar el trabajo de cálculo, esta envía al proceso encargado la linea inicial y la linea final de este, y si no es el mismo proceso maestro, también el buffer de resultados, esto permite al maestro conocer cual es el tamaño de la franja y su posición, pudiendo así introducir tales resultados en la matriz de resultados para su volcado en la memoria. En el caso especial del proceso maestro, no se requiere el envío del buffer de resultados (ni su recepción), ya que estos ya se encuentran en la matriz de resultados, ahorrando así comunicaciones innecesarias.

El hilo de recepción de resultados, en el proceso maestro, cada vez que recibe un resultado comprueba cual es el proceso que se lo ha enviado, y si el remitente no es él mismo, recibe el buffer de resultados y lo integra en la matriz de resultados.

Después de esta puesta en común de los resultados, según la estrategia de reparto, los procesos o bien terminan su ejecución, si sólo han de calcular un único segmento de la imagen como en el reparto proporcional a la velocidad o en el equitativo, o bien vuelven a solicitar, calcular y enviar más trabajos, hasta que reciben un trabajo vacío (tamaño negativo o nulo), tras lo cual termina su ejecución.

4.2.4 Implementación

El motor del trazador de rayos ALEPH se encuentra escrito en el lenguaje de programación C++, y presentaba el código necesario para realizar cálculos tanto en CPU, mediante uso de hilos de la biblioteca estándar; como en GPU, implementados en CUDA. Pero no presentaba capacidad de cálculos de forma distribuida a través de red, debido a ello, con el fin de usar dicha funcionalidad para este estudio, esta se ha tenido que desarrollar.

Este código está dividido en tres bloques:

- Base: clases que definen el nivel más bajo del motor, con inicializaciones, funciones básicas e interfaces de comunicación con el sistema.
- Core: clases que consisten el motor de manejo de objetos, formatos de salida de imagen.
- Render: clases de gestión del cálculo, generación de escenas, manejo de la cámara.

El trabajo de implementación de la funcionalidad de cálculo en paralelo distribuido por red se ha llevado a cabo en el bloque *render*, dentro de este, en el código original del motor, se hallaba una clase encargada de lanzar los cálculos y gestionar tiempos, resultados etc. llamada `Manager`.

Aprovechando el mecanismo de herencia del lenguaje C++, se ha creado una clase `MultiHostManager`, que hereda de la propia clase `Manager`. El uso de esta técnica permite reutilizar el código ya existente, y solo desarrollar los métodos y propiedades que son necesarias específicamente para el cálculo distribuido. Esta clase es la que contiene los métodos descritos en el punto anterior.

A continuación se muestra una explicación de la codificación de los métodos más relevantes para la paralelización del motor, y su esquema en pseudocódigo. La implementación real en C++ se puede consultar en el apéndice de esta memoria correspondiente al código fuente.

El método `MultiHostManager::workReceiver(FrameBuffer& f)` define el hilo de recepción de resultados y su estructura consiste en un bucle que queda a la espera de mensajes MPI etiquetados como resultados, provinientes de cualquiera de los nodos integrantes del comunicador de MPI. Al recibir uno de estos mensajes, este lleva las acciones oportunas para integrar el resultado en la matriz final.

```
desde 0 hasta numero_de_fragmentos
{
  inicio_final = recibir_de_cualquier_proceso(ETIQUETA_FRAGMENTO_A_JUNTAR,remitente);
  si es_proceso_esclavo(remitente);
  {
    buffer = recibir(ETIQUETA_MATRIZ_RESULTADOS,remitente);
    inicio = inicio_final[0];
    final = inicio_final[1];
    copiar(buffer,matriz_de_resultados,inicio,final);
}
```

`MultiHostManager::join(FrameBuffer& f)` es la parte que se encuentra enfrente del método del hilo de recepción de resultados, es decir, es el método mediante el que cada nodo envía sus resultados. Su código es simplemente un envío de mensaje MPI con los

límites del trabajo con el proceso maestro como destinatario, y un envío con el buffer de resultados, en caso de que el propio proceso no sea el nodo maestro.

```
enviar(inicio_final,maestro,ETIQUETA_FRAGMENTO_A_JUNTAR);
si soy_proceso_esclavo
{
  buffer = generar_buffer_para_enviar(imagen);
  enviar(buffer,maestro,ETIQUETA_MATRIZ_RESULTADOS);
}
```

El método `MultiHostManager::queueManager(int h)` contiene el código del hilo que hace de director en la estrategia de reparto por cola de trabajo. Es un bucle que recibe mensajes MPI solicitando un nuevo trabajo de cálculo, al que responde con una tupla que contiene el inicio y el final de la franja que el nodo esclavo debe de calcular.

Una vez se ha llegado a la altura completa de la imagen (y por tanto, se han repartido todos los trabajos), se envía un trabajo erróneo, haciendo uso de la técnica de la píldora envenenada, y los procesos interpretan como que ya no existen más trabajos.

```
particiones = partir_bien(altura,fragmentos);
desde 0 hasta fragmentos
{
  inicio_final = { particiones[i], particiones[i+1]-1 };
  recibir_de_cualquier_proceso(ETIQUETA_PETICION,demandante);
  enviar(inicio_final,demandante,ETIQUETA_INICIO);
}
veneno = { -1,-1 };
desde 1 hasta numero_de_procesos
{
  recibir_de_cualquier_proceso(ETIQUETA_PETICION,demandante);
  enviar(veneno,demandante,ETIQUETA_INICIO);
}
```

Con el objetivo de realizar una partición de la mejor manera posible, y debido a los problemas acarreados por realizar la división de manera entera, se ha diseñado la función que en este pseudocódigo se ha llamado *partir_bien* y que está definida en otra clase (base/math.h), pero por el interés que tiene debido al reflejo de la solución adoptada, se considera comentarla en este capítulo. Concretamente hace una gestión de los restos de un cociente, logrando que aunque las divisiones no logren ser exactamente iguales, sean lo más semejante posible.

```
lk = n/k;
lx = n mod k;
inicializar(s, lk);
```

```
sumar_desde_hasta(s,0,lx,1);
devuelve suma_parcial(s);
```

El método `MultiHostManager::split(int h,bool& more)` es la principal función que se encarga del reparto de trabajo. A esta función se le llama tantas veces sea necesaria (hasta que la variable *mas* tome el valor de falso, que indica que ya no se le debe de volver a invocar, ya que todo el trabajo ya está repartido) desde el método *render*.

```
mas = false;
conmutar (estrategia_de_reparto)
{
  caso EQUITATIVO:
    {
    particiones = partir_bien(altura_total,numero_de_procesos);
    inicio = particiones[identificador_proceso];
    final = particiones[identificador_proceso+1]-1;
    mas = false;
    devuelve true;
    }
    parar;
  caso PROPORCIONAL:
    {
    si soy_proceso_maestro
    {
      para r desde 0 hasta numero_de_procesos
      {
        lr[r] = velocidades_relativas(r)*altura_total;
      }
      lw = suma_componentes(lr);
      lm = altura_total - lw;
      suma_a_todos_componentes(lr,1);
      li=suma_parcial(lr);

      para r desde 0 hasta numero_de_procesos
      {
        inicio_final = { li[r], li[r+1]-1 };
        si es_proceso_esclavo(r)
        {
          enviar(inicio_final,r,ETIQUETA_INICIO);
        }
        si_no
        {
```

```
          inicio=inicio_final[0];
          final=inicio_final[1];
        }
      }
    }
    si_no //soy proceso esclavo
    {
      inicio_final = recibir(ETIQUETA_INICIO,maestro);
    }
    mas = false;
    devuelve true;
    }
    parar;
  caso COLA_DE_TRABAJOS:
    {
    enviar(maestro,ETIQUETA_PETICION);
    inicio_final = recibir(ETIQUETA_INICIO,maestro);
    si inicio_final[0]<0
    {
      mas = false;
      devuelve false;
    }
    inicio = inicio_final[0];
    final = inicio_final[1];
    mas = true;
    devuelve true;
    }
    parar;
}
devuelve false;
```

La función `MultiHostManager::render(const Camera& c,FrameBuffer& f)` es la que lanza los hilos de ejecución hacia las unidades de cálculo del propio nodo, marcando los límites que obtiene de la llamada a la función split.

También se encarga de arrancar y parar los hilos de manejo de cola de trabajos y recepción de los mismos en el nodo maestro.

```
arrancar(temporizador_maestro);
set_speeds();
si soy_proceso_maestro
{
```

```
  arranca_hilo_concurrente(workReceiver);
  si estrategia_de_reparto=COLA_DE_TRABAJOS
  {
    arranca_hilo_concurrente(queueManager);
  }
}
arrancar(temporizador);
mientras que split(altura_total,mas)=true
{
  h = final-inicio+1;
  w = ancho(imagen);
  is = inicio;
  ie = final-1;
  para cada unidad en vector_unidades
  {
    is = ie + 1;
    ie = is + int(res[i]*(h-1));
    si (ie > (inicio+h-1))
    {
      ie = inicio+h-1;
    }
    arrancar_hilo_en_unidad(unidad,is,ie,0,w-1);
  }
  esperar_hilos_en_unidades();
  join(f);
  si !mas  parar;
}
congelar(temporizador);
si soy_proceso_maestro
{
  si estrategia_de_reparto=COLA_DE_TRABAJOS
  {
    esperar_hilo(queueManager);
  }
  esperar_hilo(workReceiver);
}
congelar(temporizador_maestro);
stats(temporizador,temporizador_maestro);
```

En la función `MultiHostManager::set_speeds()` se realiza el intercambio de datos de velocidad entre los nodos, haciendo que el maestro conozca las velocidades del resto.

```
si soy_proceso_maestro
{
  velocidades[identificador_de_proceso] = mi_velocidad;
  desde 1 hasta numero_de_procesos
  {
    velocidad = recibir_de_cualquier_proceso(ETIQUETA_VELOCIDAD,remitente);
    velocidades[remitente] = velocidad;
  }
  velocidad_total = suma_componentes(velocidades);
  para cada vel en velocidades_relativas
  {
    vel = velocidades[r]/velocidad_total;
  }
  }
si_no  //soy proceso esclavo
{
  enviar(mi_velocidad,maestro,ETIQUETA_VELOCIDAD);
}
```

5 Pruebas y experimentos

Con el objetivo de medir y analizar el rendimiento así como validar las características y el funcionamiento del sistema se han seleccionado una serie de escenas y entornos de ejecución, de tal forma que podamos extraer conclusiones de la comparación de los resultados medidos en las distintas combinaciones de estos.

5.1 Escenas

Existen escenas típicas que, por sus características, por el conocimiento que se tiene sobre ellas, o simplemente por razones históricas, se usan tradicionalmente para realizar pruebas sobre sistemas de generación de imágenes por computador, ya sea como en este caso, de manera similar a la que la generaría la luz natural, o bien mediante otros procedimientos. Se ha considerado que alguna de estas escenas eran las indicadas para poner a prueba el sistema.

5.1.1 Objeto simple

Esta escena simplemente se trata de un objeto en medio de un mapa de entorno de iluminación, la razón de su inclusión en las pruebas no es otra que estudiar la manera en que afecta, o no, la complejidad de la escena en la eficiencia del cálculo de la simulación de la interacción de la luz con los materiales.

Ilustración 4: Imagen sintetizada del objeto simple

Además, también existe una pequeña razón, más personal que técnica, y es que este tipo de

escenas simples se usaron en las primeras pruebas de las etapas preliminares del código empleado para la adaptación del software al cálculo en paralelo, con lo que parecía un poco injusto dejar fuera de este trabajo estas escenas que tan útiles (y motivadoras) fueron al comienzo, para comprobar que el trabajo iba por el buen camino.

Cabe destacar la complejidad geométrica del modelo del objeto que se sitúa en el centro de la escena, en este caso un coche, con mucho detalle y por tanto un número de triángulos elevado, por lo que el cálculo de las intersecciones tendrá un peso importante en el cálculo total.

Por otro lado, la simplicidad de la escena, ya que únicamente existe un objeto, provoca que el número de rebotes sea reducido, ya que la mayor parte de los rayos saldrán hacia afuera de la escena y no deberán de continuar siendo calculados. Por este motivo, la iluminación indirecta va a tener muy poca influencia en esta escena, al contrario que la iluminación directa, que va a suponer casi la totalidad de la iluminación de la misma.

5.1.2 Cornell box

La caja de Cornell es un test que determina la precisión de los software de síntesis de imágenes que fue creado en el *Program of Computer Graphics* de la Universidad Cornell en 1984, para un paper titulado: *Modeling the Interaction of Light Between Diffuse Surfaces.*

Se creó un modelo físico, se fotografió con una cámara CCD y se midieron con exactitud las variables de la escena, tales como: posiciones de los objetos, características del foco y de

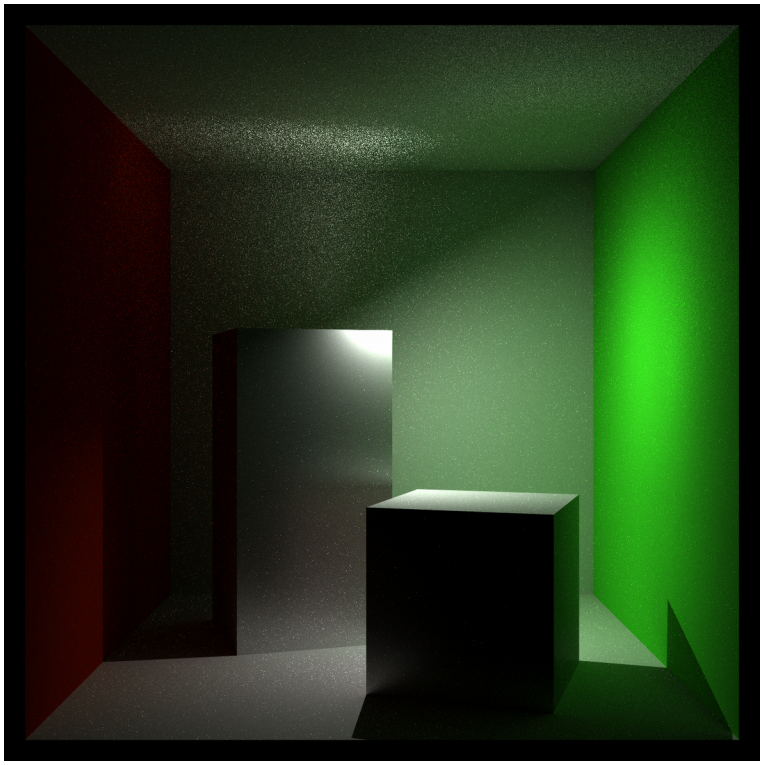


Ilustración 5: Imagen sintetizada de la caja de Cornell

los materiales etc. Con estos datos se recreó el modelo en el software de síntesis y los resultados de la simulación digital se comparaban con la fotografía.

El modelo es muy simple y consiste en una caja, donde la pared izquierda es de color rojo, la derecha verde y la de atrás blanca. Suelo y techo blancos, donde en el centro este último se encuentra el foco que ilumina la escena. En el interior de esta caja se disponen objetos como cajas, esferas, u otras que se consideren útiles para comprobar las características del software de síntesis.

La geometría de esta escena es básica y se puede modelar con un número muy bajo de triángulos. En cambio, al haber objetos de diferentes colores, son necesarias un número elevado de muestras para que el color-bleeding (cambio de color de la luz al rebotar en un objeto de color), se haga patente en la escena. El número alto de muestras también facilita que los rayos rebotados en la caja especular (caja grande), incidan sobre el techo de la caja, ya que si este número es bajo solo una parte muy pequeña de los rayos lo harían y visualmente se apreciarían puntos aislados en vez de una "mancha" de luz proyectada.

En cambio, no serían necesarios gran número de rebotes ya que la geometría es muy simple, pero aún así el peso de la iluminación indirecta es comparable al que tiene la iluminación directa.

5.1.3 Atrio del Palacio Sponza

El Palacio Sponza está situado en la ciudad croata de Dubrovnik y alberga en su interior un atrio que debido a sus pasillos perimetrales, columnas y apertura central; se trata de un escenario idóneo para experimentar con simulaciones de modelos de iluminación, sombras y rebotes.

El modelo digital de este patio permite definir determinadas posiciones de cámara donde poder observar de forma visual la validez del modelo, analizando la precisión de las sombras, o el número de reflejos necesarios para que la luz llegue a los recovecos de la escena. El disponer de varios puntos de vista también permite observar las diferencias de la complejidad de los cálculos y su eficiencia de manera analítica.

La geometría de esta escena es muy compleja y por tanto está descrita por un número muy alto de triángulos. Además, la luz incidente en la propia escena proviene de la apertura central del atrio, por lo que el peso de la iluminación directa es bastante más pequeño que el de la indirecta, que es la que ilumina los pasillos perimetrales.

Debido a ello, el número de rebotes necesarios para simular de manera correcta la iluminación en esta escena es elevado, y al ser la geometría compleja también el número de muestras tendrá que ser elevado para obtener un resultado preciso.

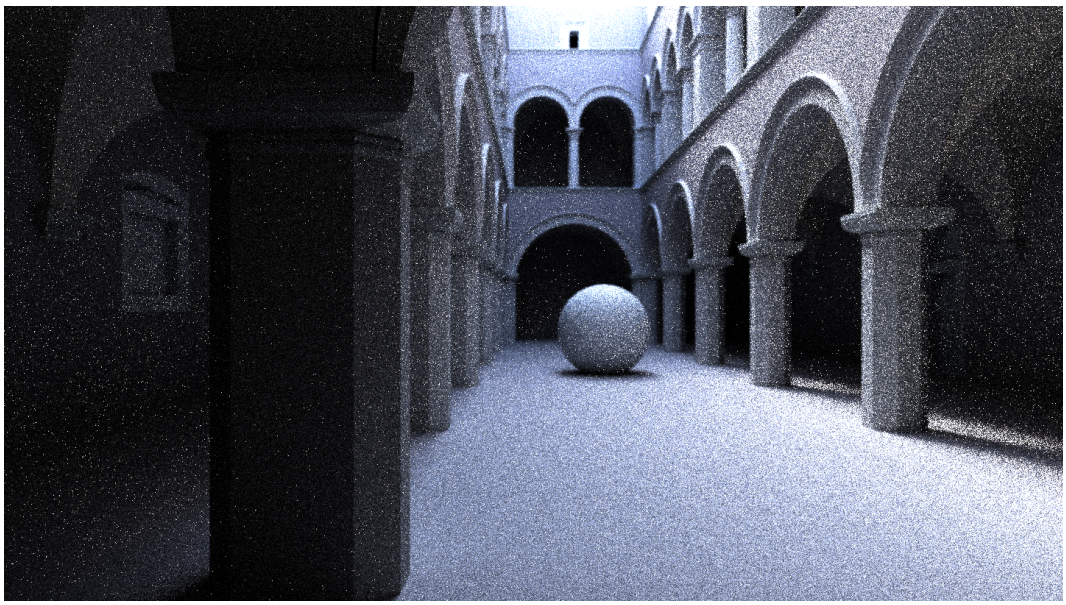


Ilustración 6: Imagen sintetizada del atrio del Palacio Sponza

5.2 Entornos de ejecución

La forma en la que está escrito el código del software trazador de rayos permite seleccionar las unidades de proceso que van a llevar a cabo el trabajo de cálculo de manera arbitraria, por lo que aunque una CPU tenga varios cores y/o hilos de proceso, se puede forzar a que solo se utilicen las necesarias, de la misma manera, si en una unidad de proceso se encuentran mas de una CPU y/o GPU, también existe la posibilidad de indicar si usar o dejar de usar estas. Adicionalmente, y ya en la capa de computación distribuida, evidentemente es posible especificar que máquinas o nodos van a formar parte de este sistema. Esto nos da tres niveles de selección de las unidades que finalmente llevarán a cabo el cálculo, y por tanto permiten definir los entornos de ejecución que se usan para realizar las pruebas.

Otro aspecto a tener en cuenta es que al utilizar las bibliotecas de MPI en el código, para que los programas hagan uso de las funciones de comunicación de esa biblioteca, se deben lanzar con las herramientas de OpenMPI destinadas a tal efecto, y son estas las que permiten especificar qué máquinas forman parte del cluster de cálculo.

Para verificar el comportamiento del sistema, y disponiendo de las máquinas ya descritas en el apartado correspondiente al hardware, se han definido los siguientes entornos:

- 1 equipo del laboratorio, limitado a 1 solo núcleo.
- 1 equipo del laboratorio, CPU completa.
- 4, 8, o 16 equipos del laboratorio.
- Annwn + Beatrix + Amelia + Grendel + Wargo, o cada una de ellas por separado.

- Annwn + Beatrix + Amelia + Grendel + Wargo + Lugh + Bran.

5.3 Parámetros

La síntesis de imágenes puede depender de gran cantidad de variables, las cuales se pueden ajustar para escalar la complejidad (y exactitud) de la imagen a generar. Con esto se consigue variar el número de cálculos necesario para generar la imagen, y de forma consecuente, el tiempo necesario para calcularla.

Los parámetros principales a ajustar serían: el tamaño de la imagen que denotamos con un multiplicador 'k', la profundidad del cálculo (número de rebotes de un mismo rayo, de forma que se simula la luz reflejada) 'b', o el número de muestras calculadas (rayos) para cada píxel de la imagen 's'.

Los parámetros usados para cada escena son:

- Objeto simple: k=8 b=8 s=256 que da como resultado una imagen cuadrada de 1080 píxeles de lado.
- Cornell Box: k=8 b=3 s=256 que también tiene como resultado una imagen cuadrada de 1080 píxeles de lado.
- Palacio Sponza: k=8 b=4 s=256 que da como resultado una imagen de tamaño 1920 píxeles de ancho y 1080 píxeles de alto.

La razón de la elección de estos parámetros y no otros cualesquiera únicamente radica en que se buscaba conseguir una imagen que tuviera carga importante de trabajo, pero que permitiese hacer varias pruebas a lo largo del día, evitando así cálculos de excesiva duración que además no aportarían valor adicional al estudio, ya que los tiempos obtenidos ya permiten extraer datos lo suficientemente relevantes como para sacar conclusiones al respecto.

Adicionalmente, la estrategia de reparto de trabajo a emplear también se trata de un parámetro ajustable, y que en el caso de la cola de trabajo, introduce otra variable, el número de fragmentos en que se divide la imagen. Como la teoría nos dice que el reparto de trabajo influye directamente en el rendimiento del sistema resulta interesante estudiar varios casos, y debido a ello no se han definido fijos por escena, sino que se han aplicado varios valores para ellos en todas las escenas.

En este aspecto cabe destacar que las estrategias de cola de trabajos y equitativa se tratan esencialmente de la misma cuando el número de segmentos en los que se divide coincide con el número de nodos del sistema. También, aunque se ha implementado la estrategia de reparto proporcional a la velocidad, esta prácticamente coincide con la equitativa en los casos en que las máquinas tienen características similares, debido a que el motor está en

desarrollo y no dispone de los recursos para medir la velocidad efectiva de cada máquina, teniendo en cuenta solo el número de unidades de proceso de cada máquina y no las capacidades reales de estas.

5.4 Pruebas

Con la realización de las pruebas, además de validar el correcto funcionamiento del sistema, se ha buscado estudiar aspectos propios de la paralelización, de las máquinas o del propio algoritmo.

Concretamente, se ha estudiado el comportamiento del sistema con respecto a los siguientes aspectos.

- Paralelización a nivel de CPU
- Paralelización a nivel de nodo
- Paralelización en un sistema homogéneo
- Comparación de estrategias de reparto
- Paralelización en un sistema heterogéneo
- Influencia de la escena

5.4.1 Medidas de rendimiento

5.4.1.1 Eficiencia

Se define la *eficiencia* conseguida mediante la siguiente fórmula:

$$E=\frac{T_1}{T_n \cdot n}$$

Donde n es el número de unidades de proceso, y T_n el tiempo empleado para calcular la imagen completa con n unidades.

El dato de la eficiencia no resulta tan sencillo de calcular en los sistemas heterogéneos donde las computadoras no son similares en cuanto a capacidad de cálculo, debido a que no se cuenta con un tiempo de referencia que pueda ser tomado para sustituir al tiempo que en el caso del cluster homogéneo era del de una máquina.

Si se desarrolla el cálculo de la eficiencia para el caso de máquinas similares:

$E=\frac{T_1}{T_n \cdot n}=\frac{T_1/n}{T_n}$, y se define $T_r=T_1/n$ que denota el tiempo ideal en paralelo.

Tomando la potencia de cada máquina, P_i , siendo el trabajo W , como:

$$P_i = \frac{W}{T_i}$$

Se puede escribir T_r , de la siguiente manera:

$$T_r = \frac{W}{P_n} = \frac{W}{\sum_1^n P_i} = \frac{W}{\sum_1^n \frac{W}{T_i}} = \frac{1}{\sum_1^n \frac{1}{T_i}}$$

Dado que la media armónica, H , que siendo n el número de máquinas, y x_i el tiempo empleado por la máquina i está definida como:

$$H = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}} \quad \text{y dado que} \quad T_r = \frac{H}{n} \quad ,$$

se puede usar H para sustituir el tiempo de referencia, y entonces, la eficiencia para estos casos se toma como:

$$E = \frac{T_r}{T_n} = \frac{H/n}{T_n} = \frac{H}{T_n \cdot n}$$

5.4.1.2 Balanceo de carga

Es interesante introducir un dato estadístico que pueda dar idea de como de balanceada está la carga de trabajo entre las unidades. Se ha optado por usar la *desviación típica* que denotaremos con σ y que, siendo n el número de muestras, x_i la propia muestra y $\bar{x}$ la media aritmética de todas las muestras, se define como:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^n (x_i - \bar{x})^2}$$

Para poder comparar esta magnitud de forma independiente a los tiempos en las distintas pruebas, ya que estos pueden ser muy diferentes, se puede recurrir a escalarla dividiéndola entre la media aritmética $\bar{x}$ de las muestras, lo que resulta en un factor entre 0 y 1, que denominaremos *factor de desbalanceo*, y notaremos como δ . Al restar de la unidad esta magnitud, obtendremos su complementario, que nos dará información sobre el equilibrio de la carga de trabajo, y le denominaremos *factor de balanceo*, β .

$$\delta = \frac{\sigma}{\bar{x}} \qquad 1 - \delta = \beta$$

Esta magnitud puede tomar valores entre 0 y 1, y cuanto más cercana sea a 1, menos dispersión existirá entre las muestras ya que estas tendrán menos diferencias entre sí, y por tanto mejor balanceado estará el trabajo entre las unidades.

El valor de este dato se calcula en tiempo de ejecución del programa trazador de rayos.

6 Resultados

6.1 CPU

Todas las CPU de las computadoras que han formado parte de este estudio poseen varios núcleos, con lo que es posible realizar cálculos de manera paralela en la propia CPU.

Se han tomado como referencia los tiempos obtenidos por una computadora del laboratorio 0.01 del edificio Ada Byron de la EINA.

En esta prueba se ha sintetizado la imágenes de las escenas de prueba y se han medido los tiempos con un único núcleo de la CPU, y con todos ellos (la CPU usada en esta prueba dispone de 4 núcleos).

La siguiente tabla muestra el tiempo, en milisegundos y la eficiencia conseguida con una CPU para las tres escenas de prueba:

Escena	Núcleos	Tiempo	Eficiencia
Simple	1	386631	1,00
	4	133605	0,72
Cbox	1	1219451	1,00
	4	418904	0,73
Sponza	1	6875196	1,00
	4	2320158	0,74

Tabla 2: Resultados en función del número de núcleos en una misma CPU

Se puede apreciar que la eficiencia alcanzada con los 4 núcleos en paralelo se queda en un valor alrededor de 0.75.

Este fenómeno puede estar debido a las colisiones de caché, ya que al calcularse distintas áreas de la imagen de manera concurrente la caché puede aprovechar menos la localidad de los datos que si se calculase de manera secuencial.

6.2 GPU – CPU

Continuando con las pruebas y verificaciones a nivel de nodo, nos encontramos que las máquinas que cuentan con GPUs tienen capacidad de realizar cálculos tanto con sus CPUs como con sus GPUs al mismo tiempo, con lo que resulta interesante analizar que rendimiento se obtiene al ponerlas a trabajar a la vez.

En esta prueba, se ha calculado la imagen del palacio Sponza en las máquinas que disponen de GPU, se han medido los tiempos de cálculo y el balanceo que medimos mediante β .

Máquina	Tiempo	β
Beatrix	1208840	0,918
Amelia	1394664	0,882
Grendel	3325563	0,890
Wargo	3293219	0,891

Tabla 3: Resultados en función de la máquina con el balanceo CPU-GPU

El desequilibrio que se aprecia puede tener varias lecturas: una de ellas es que puede deberse a la problemática descrita en el punto donde se describe la paralelización a nivel de nodo, es decir, la limitada capacidad para comparar las velocidades relativas de una unidad de proceso frente a otra (CPU frente a GPU).

Por otro lado, debido a la experimentalidad del código, y que este se encuentra en una fase de desarrollo, todavía hay segmentos en el código de las funciones donde la GPU realiza el trabajo de cálculo, que no se encuentran migrados a CUDA, y que se tienen que apoyar en la capacidad de la CPU para poder llevarse a cabo. El más significativo de estos segmentos de código está en la propia evaluación de la función de Monte Carlo que obtiene la dirección del rayo rebotado en una superficie.

6.3 Cluster homogéneo

A partir de esta prueba es donde se procede a validar el entorno distribuido por red. Aunque más adelante en este estudio se analiza el comportamiento del cluster heterogéneo (con máquinas de diferentes capacidades), se ha querido dedicar un apartado a validar las capacidades de computación masivamente paralela realizando las pruebas con una cantidad considerable de máquinas, por ello, se ha configurado un cluster con los equipos del laboratorio 0.01 del edificio Ada Byron de la EINA.

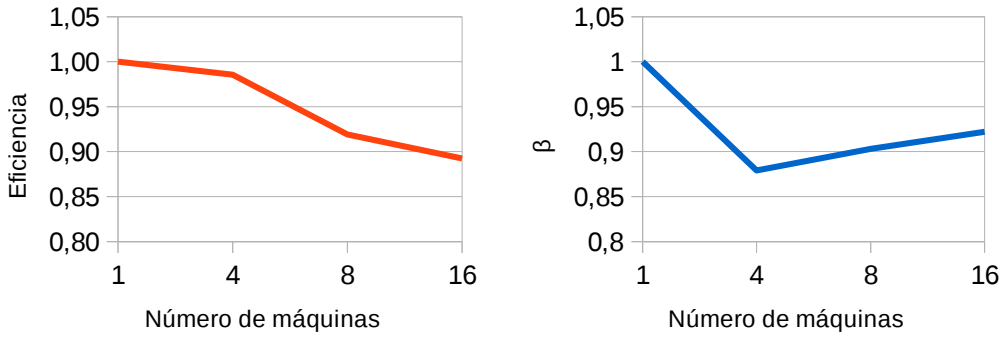
Estas computadoras cuentan con una CPU de 4 núcleos, y no disponen de GPU. Todas estas máquinas son de idénticas características y por ello, para este caso, denominaremos esta formación como cluster homogéneo. Intuitivamente se puede pensar que al tratarse de máquinas iguales el reparto de la carga de trabajo resulta sencillo pero la solución a este problema veremos que no resulta tan evidente.

Se ha querido usar este cluster homogéneo con varios objetivos: en una mano validar la escalabilidad del sistema, analizando el rendimiento conseguido al añadir nodos al sistema; y en la otra, verificar las diferencias entre las diferentes estrategias de reparto de trabajo.

A tal fin se ha lanzado el cálculo de la escena del palacio Sponza con 1, 4, 8 y 16 máquinas del laboratorio 0.01, con las estrategias de reparto equitativa y cola de trabajos (con un número de trabajos fijado a 80). La razón de no incluir en esta prueba la estrategia proporcional a la capacidad se debe a que al ser las máquinas de similares características el reparto realizado por dicha estrategia seria el mismo que el que realiza la equitativa.

Máquinas	Tiempo	Eficiencia	β
1	2320158	1,00	1
4	588651	0,99	0,879
8	315513	0,92	0,903
16	162500	0,89	0,922

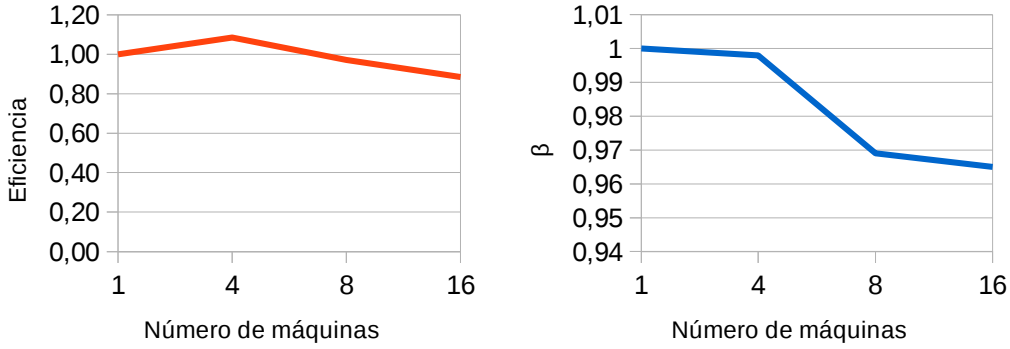
Tabla 4: Resultados en función del número de máquinas para la estrategia de raparto equitativa



Estos resultados son los esperados, ya que aumentando el número de máquinas, aunque el balanceo se mantiene en valores alrededor de 0.90, la eficiencia se ve disminuida. La razón de esta pérdida de eficiencia puede explicarse por la razón de que son necesarias más comunicaciones, y además, el tiempo dedicado a estas es relativamente mayor al del cálculo que con un menor número de máquinas, ya que el tamaño del fragmento a calcular por cada máquina en la estrategia de reparto equitativa es inversamente proporcional al número de máquinas.

Máquinas	Tiempo	Eficiencia	β
1	2320158	1,00	1
4	534979	1,08	0,998
8	298794	0,97	0,969
16	163933	0,88	0,965

Tabla 5: Resultados en función del número de máquinas para la estrategia de reparto por cola de trabajos con 80 fragmentos



Con 80 fragmentos en la cola de trabajos, la eficiencia llega hasta ser mayor que la unidad, esto, aparentemente imposible, puede explicarse mediante el funcionamiento correcto de la memoria caché de las CPU, que hacen uso de la localidad de datos pero son limitadas en capacidad, por lo que deben ir cargándose y descargándose para contener a los espacios de memoria con los que trabajar, en cambio, al estar replicado y esta caché no ser compartida, este intercambio de información con la memoria principal no es necesario. Aún así, este fenómeno no se manifiesta con tanta intensidad cuando el número de máquinas se incrementa, ya que, como en el caso del reparto equitativo, las comunicaciones pasan a tener más peso en el tiempo total y por consiguiente, la eficiencia del sistema cae.

Con respecto al balanceo se aprecia que es prácticamente idóneo aunque decrece ligeramente cuando aumenta el número de máquinas, esto principalmente se debe a la diferencia de carga entre unos fragmentos y otros de la imagen.

Comparando ambas estrategias de reparto se ve una considerable mejora del tiempo conseguida con la cola de trabajos, de mayor manera cuando existen más fragmentos por máquina, ya que de esta manera el sistema se autobalancea mucho mejor.

6.3.1 Cola de trabajos

Tomando las conclusiones del punto anterior, y viendo las ventajas de la estrategia de reparto mediante cola de trabajos, es preciso realizar la comparativa con el parámetro ajustable de este método de reparto, el número de trabajos o fragmentos en los que se divide la imagen.

En esta prueba se ha lanzado el cálculo de la escena del palacio Sponza contra el cluster homogéneo de 16 máquinas del laboratorio 0.01, variando el número de trabajos entre 12, 40, 160 y 280.

Fragmentos	Tiempo	Eficiencia	β
12	165631	0,88	0,919
40	147267	0,98	0,865
80	163933	0,88	0,965
160	150856	0,96	0,976
280	128948	1,12	0,989

Tabla 6: Resultados en función del número de fragmentos para la estrategia de reparto por cola de trabajos

Para la confección de la tabla, como tiempo de referencia para el cálculo de la eficiencia se ha tomado el de una sola máquina (2320158 ms.).

Configurando el número de fragmentos a 12, siendo el cluster de 16 máquinas se ha querido forzar el que hubiera máquinas sin trabajos para realizar, y de hecho esto se pone de manifiesto al conseguir de esta manera una eficiencia más baja que las normalmente alcanzadas por esta estrategia de reparto.

Las demás configuraciones, ponen de manifiesto la importancia de la elección de un número de fragmentos razonable y consecuente tanto con el número de máquinas que componen el clúster como con el propio tamaño de la imagen, pues la eficiencia conseguida puede depender en gran medida de esta elección.

6.4 Cluster heterogéneo

El objetivo principal de este estudio es crear un sistema de trazado de rayos sobre un cluster heterogéneo, donde las computadoras que formen parte de él contengan CPUs y GPUs. Para ello, se ha configurado el cluster para hacer uso de 5 máquinas situadas en los laboratorios del GIGA: Amelia, Annwn, Beatrix, Grendel y Wargo, que han sido presentadas junto a sus características en la sección donde se describe el sistema.

Al existir diferencias de características y capacidades entre las máquinas, en esta prueba toma aún más relevancia el dato del factor de balanceo, pues va a dar una visión de la capacidad de adaptación de cada estrategia de reparto, y del aprovechamiento de la capacidad total de cálculo del sistema completo.

La prueba se trata de la síntesis de la imagen de la escena del palacio Sponza, con las distintas estrategias de reparto: equitativa, proporcional a la capacidad y cola de trabajos. En el caso del reparto por cola de trabajos, se ha fijado el número de fragmentos en 80.

Máquina	Tiempo
Beatrix	1208840
Amelia	1394664
Grendel	3325563
Wargo	3293219
Annwn	1368327
M. Armónica	1736453,2121

Tabla 7: Tiempos por nodo del C. heterogéneo

Estrategia	Tiempo	Eficiencia	β
Equitativa	631268	0,55	0.727
Proporcional	553753	0,63	0.812
Cola trabaj.	400479	0,87	0.974

Tabla 8: Resultados para el cluster heterogéneo en función de la estrategia de reparto

Analizando los datos de esta prueba llama la atención la baja eficiencia conseguida con el reparto equitativo, y es de esperar, debido a la diferencia de rendimiento entre máquinas y a la diversidad de carga de trabajo de los fragmentos de la imagen: dependiendo de a qué máquina le toque procesar según que fragmento esto puede suponer un desbalanceo todavía mayor que el propio generado por las diferencias entre las máquinas. La diferencia de tiempos para el trabajo completo entre el más rápido, Beatrix, y el más lento, Grendel, es de 2.75 veces mayor para el más lento.

Para aliviar este desbalanceo, se opta por probar las otras estrategias de raparto de trabajo, y se verifica en los resultados, que la estrategia que consigue mejor equilibrio y también más eficiencia es la del reparto por cola de trabajos.

6.4.1 Desbalanceado

Como prueba adicional, y para hacer el cluster más desbalanceado, al cluster heterogéneo de las máquinas del GIGA, se le han añadido dos nodos: lugh y bran, con una capacidad mucho más limitada que los del punto anterior. Para evidenciar esta diferencia, disponemos de los datos de los tiempos que invierte en el cálculo la computadora que más rápido ha obtenido el resultado: Beatrix, en 1208840 ms., mientras que la mas lenta, lugh, ha invertido 28011752 ms., nótese que el tiempo de lugh es más de 23 veces mayor que el de Beatrix.

Con este deajuste se ha buscado probar el impacto que tiene la coordinación entre los nodos, y como varía dependiendo de la estrategia de reparto.

Máquina	Tiempo
Beatrix	1208840
Amelia	1394664
Grendel	3325563
Wargo	3293219
Annwn	1368327
Lugh	28011752
Bran	27764628
M. Armónica	2371957,5309

Tabla 9: Tiempos por nodo del C. heterogéneo desbalanceado

Estrategia	Tiempo	Eficiencia	β
Equitativa	2726194	0,12	0.610
Proporcional	701738	0,48	0.742
Cola trabaj.	505869	0,67	0.875

Tabla 10: Resultados para el cluster desbalanceado en función de la estrategia de reparto

En estos resultados se pueden apreciar los mismos efectos que con el cluster del punto anterior, pero mucho más acusados, debido a la extrema diferencia de potencias entre la computadora más rápida y la más lenta. Se ponen especialmente de manifiesto con el uso de la estrategia de reparto equitativa, ya que las máquinas más potentes deben de esperar a que las más lentas terminen de procesar el fragmento que se les ha asignado.

Aunque en esta prueba se ha fijado el número de fragmentos a 80 con el objetivo de poder comparar los dos cústeres con los mismos parámetros en las pruebas, se puede prever que con un mayor número de fragmentos, el balanceo se habría realizado de mejor manera y la eficiencia hubiera sido mayor. Teóricamente, dada la diferencia de 23:1. es previsible que un número de fragmentos de 23 * 7, debido a la diferencia y al número de máquinas daría mejores resultados, ya que la máquina más veloz realizaría 23 trabajos en el tiempo que la más lenta haría 1.

6.5 Escenas

Es interesante incidir en el hecho de que en el equilibrio del reparto del trabajo no solo influye la capacidad de los nodos, o de las unidades de cálculo de las que disponen, sino que otro factor a tener en cuenta es la propia complejidad de la escena.

Esta complejidad, puede no estar repartida uniformemente en todo el área de la imagen, y según la estrategia de reparto utilizada, o las capacidades de la máquina a la que se asigne según que fragmento de imagen, los resultados pueden variar.

En esta prueba se ha lanzado el cálculo de las tres escenas, sobre el cluster heterogéneo y sobre el cluster homogéneo, usando la estrategia de reparto por cola de trabajos, con un número de trabajos de 80.

	Cluster homogéneo			Cluster heterogéneo		
Escena	Tiempo	Eficiencia	β	Tiempo	Eficiencia	β
Simple	8326	1,00	0.979	58384	0,51	0.870
Cbox	23532	1,11	0.953	97053	0,78	0.970
Sponza	163933	0,88	0.965	400479	0,87	0.974

Tabla 11: Resultados en función de las escenas para los clusteres homogéneo y heterogéneo

7 Conclusiones y trabajo futuro

7.1 Cumplimiento de objetivos

El objetivo principal de este proyecto, que consistía en la adaptación del software de simulación de la interacción luz-material al cálculo en paralelo y el estudio de su comportamiento.

Se ha implementado una solución basada en MPI para dotar al motor trazador de rayos ALEPH/FTL, la capacidad de realizar cálculos de manera concurrente en varios equipos conectados por red.

Se ha estudiado la problemática del reparto de trabajo en un cluster que no tiene por qué estar balanceado, proponiendo tres estrategias de reparto: equitativa, proporcional a la capacidad y mediante cola de trabajos.

Una vez propuestas, se han comparado, quedando evidenciada la mejora de rendimiento y aprovechamiento de la capacidad potencial del sistema mediante la estrategia de cola de trabajos.

El objetivo secundario consistente en la aportación del código para esta adaptación al motor de renderizado ALEPH desarrollado en el GIGA, queda validado también con los resultados anteriores, quedando disponible en el repositorio de código de tal proyecto para futuras revisiones y adaptaciones, o bien sirviendo de base para la implementación de nuevas características. Este código también abre posibilidades de estudio del código en sí mismo y de las tecnologías usadas para su desarrollo.

7.2 Problemas e incidencias

En el transcurso de la realización de este proyecto de fin de carrera han ocurrido ciertos problemas y dificultades que se han logrado solventar bien adquiriendo conocimientos técnicos, aplicando mejoras y herramientas, o adoptando soluciones fruto del ingenio del alumno o de los directores.

Los principales problemas se han debido a la forma de sincronizar recepción de trabajos, comunicación entre nodos, etc. que se han solucionado implementando nuevos métodos, o bien reestructurando los ya existentes, después de muchas pruebas erróneas y potenciales soluciones hasta que se ha llegado a las definitivas.

También se han solventado tanto problemas como tareas repetitivas con la creación de pequeños scripts que facilitaban dichas tareas, entre otras, conocer qué equipos se encontraban encendidos (o directamente encenderlos por red, por medio de *wake on lan*), para poder lanzar las simulaciones en ellos, o para encontrar la ruta precisa a las

bibliotecas y añadirlas a las variables de entorno para que el software no tuviera problemas en localizarlas, ya que debido a la heterogeneidad entre los sistemas operativos de las máquinas, estas pueden no coincidir en todos los casos.

Como en cualquier tipo de proyecto, la gestión del tiempo, disponibilidad, y recursos ha supuesto un reto que, aunque al final se ha sido capaz de controlar, ha traído retrasos y situaciones no idóneas.

7.3 Valoración del autor

La valoración de la realización de este PFC resulta muy satisfactoria, ya que ha permitido al alumno adentrarse más en el conocimiento de la programación paralela y la computación de alto rendimiento.

También le ha supuesto conocer las técnicas de simulación de la luz, tema con el que no había tratado antes, así como la física que existe detrás de todos esos cálculos que son llevados a cabo por la simulación.

Este proyecto, también ha traído consigo la oportunidad de trabajar con tecnologías propias de los mayores supercomputadores del mundo, así como solventar problemas y adquirir conocimientos que de otra manera difícilmente se hubiera dado el caso.

El alumno ha tenido que profundizar en el conocimiento de estas tecnologías, como por ejemplo OpenMPI, que aunque presentadas en algunas de las asignaturas cursadas en el transcurso de la carrera, se han tenido que adaptar al problema concreto de la síntesis de imágenes.

Otro aspecto a valorar es el reto que ha supuesto a la capacidad del alumno al enfrentarse a nuevos problemas, a demostrar y a aplicar los conocimientos adquiridos durante la carrera.

7.4 Trabajo futuro

Al tratarse de un software experimental y en fases iniciales de desarrollo, este cuenta con muchas posibles mejoras, tanto en el software que se ha tomado como base, como en el perfeccionamiento del código para la paralelización que se ha aportado mediante la realización de este proyecto.

Un aspecto interesante a mejorar en la paralelización es la capacidad de estimar la velocidad de una determinada máquina, probablemente con un microbenchmark al arrancar el software, con una medida de la velocidad fiable, se optimizaría el reparto de trabajo proporcional a la velocidad.

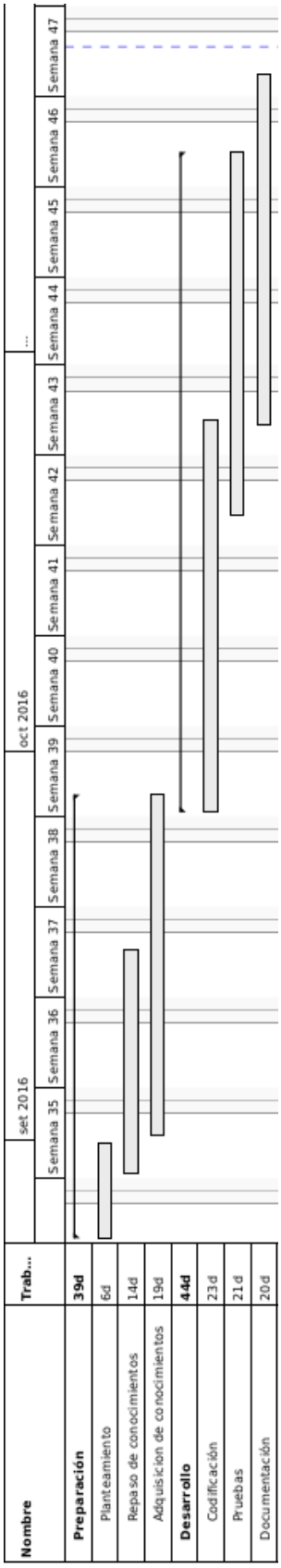
Gracias a la batería de pruebas llevadas a cabo en este estudio, se ha evidenciado el problema de reparto entre las unidades de un nodo (CPU – GPU), por lo que se ha abierto

un camino en el desarrollo del propio motor para implementar un reparto de trabajo mediante cola de trabajos también a este nivel.

También se encuentra en vías de desarrollo la migración del cálculo de un rayo de forma completa en la GPU, y esto supondrá el aislamiento completo de los cálculos que se realizan en la GPU con respecto a los que se hacen en la CPU, lo que hará que esos cálculos sean independientes y que el rendimiento no se vea afectado por las otras unidades de proceso de la misma máquina.

8 Diagrama Temporal

El trabajo de este proyecto de fin de carrera se ha estimado en 650 horas, repartidas según el esquema temporal.



1 Apéndice: Referencias

[CGIB86] Cohen M.F., Greenberg D.P., Immel D.S. y Brock P.J.: An efficient radiosity approach for realistic image synthesis. IEEE Computer Graphics and Applications, tomo 6(3), págs 26-35 (1986) ISSN 0272-1716.

[CPC84] Cook R.L., Porter T. y CarpenterL.: Distributed Ray Tracing. En Computer Graphics (ACM SIGGRAPH '84 Proceedings), tomo 18, págs. 137-148 (1984).

[Amd67] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", 1967.

[Gre91] Green S.: Parallel Processing for Computer Graphics (Pitman Publishing, 1991).

[Hal88] Hall R.: Illumination and Color in Computer Generated Imagery (Springer, Berlin, 1988).

[Jen01] Jensen H.: Realistic image synthesis using photon mapping (A.K. Peters, Natick, Massachusets, 2001).

[Mag03] Magallón, J. A.: ALEPH. Simulación realista de la iluminación global mediante técnicas de MonteCarlo y Procesado paralelo. Tesis doctoral de ingeniería industrial, Universidad de Zaragoza, 2003.

[MPI] The message passing interface (mpi) standard. http://mpi-forum.org/docs/

2 Apéndice: Código fuente

Si bien resulta imposible la inclusión de todo el código fuente del trazador de rayos usado para este estudio, si que se ha estimado que resulta útil y esclarecedor hacerlo con la clase que realiza las tareas de paralelización MultiHostManager, ya que contiene el código fuente que se ha tenido que desarrollar para poder llevar a cabo este estudio.

2.1 Clase MultiHostManager

2.1.1 render/multihostmanager.h

```
#pragma once

#include <ftl/render/manager.h>
#include <ftl/base/dmp.h>

#ifdef CONFIG_HAS_MPI

namespace FTL {
namespace render {

using namespace FTL::base;
using namespace FTL::core;

enum distribution_t {EQUALLY, SPEED_PROPORTIONAL, WORK_QUEUE};

#define NODESPD_TAG    1
#define WORLDSPD_TAG  2
#define CHNKSTART_TAG  3
#define CHNKEND_TAG    4
#define CHNKTOJOIN_TAG  5
#define DATAMATRIX_TAG  6
#define WQREQUEST_TAG  7
#define WQCHNKSTART_TAG  8
#define DURATION_TAG  9

class __public_render MultiHostManager : public Manager
{
private:
    int        _dist;
    int        _chunks;
```

```
  float      _myspeed;
  vector<float>  _speed;
  vector<float>  _rspeed;
  float          _wspeed;
  vector<float>  _work;
  int myis;
  int myie;

  float myspeed();
  void set_speeds();

    void queueManager(int h);
    void workReceiver(FrameBuffer& f);
    bool split(int h,bool& more);
    void init();
    void fini();

    void join(FrameBuffer& f);

protected:
    void stats(const Duration& et, const Duration& met,
        const vector<Duration>& eng) const;

public:
    MultiHostManager(int dist=SPEED_PROPORTIONAL, int chunks = 0);
    MultiHostManager(bool uc, bool ug, int dist=SPEED_PROPORTIONAL, int chunks = 0);
    MultiHostManager(int nc,int kc,int ng,int kg, int dist=SPEED_PROPORTIONAL, int chunks
= 0);
    ~MultiHostManager();
    int chunks() const;
    void chunks(int n);

    void render(const Camera& c,FrameBuffer& f);
};
}
}

#endif
```

2.1.2 render/multihoshmanager.cc

```
#include "ftl/render/manager.h"
#include "ftl/render/multihostmanager.h"
```

```
#include "ftl/base/sysinfo.h"
#include "ftl/base/dmp.h"
#include "ftl/base/logger.h"
#include "ftl/core/tonemap.h"
#include "ftl/core/png.h"

#include "ftl/render/shader.h"

#ifdef CONFIG_HAS_MPI

namespace FTL {
namespace render {

using namespace FTL::base;
using namespace FTL::core;

MultiHostManager::MultiHostManager(int dist, int chunks)
  : Manager()
{
  _dist   = dist;
  _chunks = (_dist==EQUALLY || _dist==SPEED_PROPORTIONAL) ? DMP::csize() : chunks;
  if (_chunks<DMP::csize()) _chunks = DMP::csize();
  _myspeed = myspeed();
  _wspeed = 0;
}

MultiHostManager::MultiHostManager(bool uc, bool ug, int dist, int chunks)
  : Manager(uc,ug)
{
  _dist   = dist;
  _chunks = (_dist==EQUALLY || _dist==SPEED_PROPORTIONAL) ? DMP::csize() : chunks;
  if (_chunks<DMP::csize()) _chunks = DMP::csize();
  _myspeed = myspeed();
  _wspeed = 0;
}

MultiHostManager::MultiHostManager(int nc,int kc,int ng,int kg, int dist, int chunks)
  : Manager(nc,kc,ng,kg)
{
  _dist   = dist;
  _chunks = (_dist==EQUALLY || _dist==SPEED_PROPORTIONAL) ? DMP::csize() : chunks;
```

```
  if (_chunks<DMP::csize()) _chunks = DMP::csize();
  _myspeed = myspeed();
  _wspeed = 0;
}

MultiHostManager::~MultiHostManager()
{
  fini();
}

void MultiHostManager::fini()
{
//  logger::lock();
//  logger::cinfo() << "Process " << DMP::rank()
//          << " ending [on " << SysInfo::name() << "]" << endl;
//  logger::unlock();
}

float MultiHostManager::myspeed()
{
  float tes = 0.0f;
  for (const auto& e : engines)
    tes += e->speed();

  return tes;
}

bool MultiHostManager::split(int h,bool& more)
{
  more = false;
  MPI_Status status;
  switch (_dist)
  {
    case EQUALLY:
      {
      vector<int> lk = spliti(h,DMP::csize());
      myis = lk[DMP::rank()  ];
      myie = lk[DMP::rank()+1]-1;
      more = false;
      return true;
      }
```

```
      break;

    case SPEED_PROPORTIONAL:
      {
      if (DMP::master())
      {
        vector<int> lr(DMP::csize());
        for (int r=0; r<DMP::csize(); r++)
          lr[r] = int(_rspeed[r]*h);
        int lw = accumulate(begin(lr),end(lr),0);
        int lm = h - lw;
        for (int r=0; r<lm; r++)
          lr[r]++;
        vector<int> li(DMP::csize()+1,0);
        partial_sum(begin(lr),end(lr),begin(li)+1);
        for (int r=0; r<DMP::csize(); r++)
        {
          int ise[2] = { li[r], li[r+1]-1 };
          if (DMP::slave(r))
          {
#if 0
            MPI_Send(&ise[0],1,MPI_INT,r,CHNKSTART_TAG,MPI_COMM_WORLD);
            MPI_Send(&ise[1],1,MPI_INT,r,CHNKEND_TAG,MPI_COMM_WORLD);
#else
            MPI_Send(ise,2,MPI_INT,r,CHNKSTART_TAG,MPI_COMM_WORLD);
#endif
          }
          else
          {
            myis=ise[0];
            myie=ise[1];
          }
        }
      }
      else
      {
#if 0
        MPI_Recv(&myis,1,MPI_INT,DMP::masterid(),CHNKSTART_TAG,MPI_COMM_WORLD,&status);
        MPI_Recv(&myie,1,MPI_INT,DMP::masterid(),CHNKEND_TAG,MPI_COMM_WORLD,&status);
#else
        int ise[2];
```

```
        MPI_Recv(ise,2,MPI_INT,DMP::masterid(),CHNKSTART_TAG,MPI_COMM_WORLD,&status);
        myis=ise[0];
        myie=ise[1];
#endif
      }
      more = false;
      return true;
      }
      break;

    case WORK_QUEUE:
      {
      int buff = 0;
      MPI_Send(&buff,1,MPI_INT,DMP::masterid(),WQREQUEST_TAG,MPI_COMM_WORLD);
      int ise[2];
      MPI_Recv(&ise,2,MPI_INT,DMP::masterid(),WQCHNKSTART_TAG,MPI_COMM_WORLD,&status);
      if (ise[0]<0)
      {
        more = false;
        return false;
      }
      myis = ise[0];
      myie = ise[1];
      more = true;
      return true;
      }
      break;
  }

  return false;
}

void MultiHostManager::queueManager(int h)
{
  MPI_Status status;
  int buff=0;

  vector<int> lk = spliti(h,_chunks);
  for (int i=0; i<_chunks; i++)
  {
    int ise[2] = { lk[i], lk[i+1]-1 };
```

```
    MPI_Recv(&buff,1,MPI_INT,MPI_ANY_SOURCE,WQREQUEST_TAG,MPI_COMM_WORLD,&status);
    MPI_Send(&ise,2,MPI_INT,status.MPI_SOURCE,WQCHNKSTART_TAG,MPI_COMM_WORLD);
  }
  logger::lock();
  logger::cinfo() << "QueueMGR: No jobs left" << endl;
  logger::unlock();
  static const int done[2] = { -1,-1 };
  for (int i=0; i<DMP::csize(); i++)
  {
    MPI_Recv(&buff,1,MPI_INT,MPI_ANY_SOURCE,WQREQUEST_TAG,MPI_COMM_WORLD,&status);
    MPI_Send(&done,2,MPI_INT,status.MPI_SOURCE,WQCHNKSTART_TAG,MPI_COMM_WORLD);
  }
}

void MultiHostManager::workReceiver(FrameBuffer& f)
{
//  f.blank();

  _work.assign(DMP::csize(),0.0f);

  for (int n=0; n<_chunks; n++)
  {
    MPI_Status status;
    int ise[2];
    MPI_Recv(&ise,2,MPI_INT,MPI_ANY_SOURCE,CHNKTOJOIN_TAG,MPI_COMM_WORLD,&status);
    int is = ise[0];
    int ie = ise[1];
    int nr = ie-is+1;
    _work[status.MPI_SOURCE] += nr;
    if (DMP::slave(status.MPI_SOURCE))
    {
      int sz = nr*f.width()*SDF::nsamples;

      float* buffer = new float[sz];


MPI_Recv(buffer,sz,MPI_FLOAT,status.MPI_SOURCE,DATAMATRIX_TAG,MPI_COMM_WORLD,&status);
      // row
      for (int i=0; i<nr; i++)
      {
        // col
```

```
        vector<SDF>& row = f[is+i];
        for (int j=0; j<f.width(); j++)
        {
          float* buf = buffer + (i*f.width()+j)*SDF::nsamples;
          row[j] = buf;
        }
      }
      delete[] buffer;
    }
#if 0
    logger::lock();
    logger::cinfo() << "Buf received [" << n << "/" << realChunks << "]" << endl;
    logger::unlock();
#endif
  }

  for (auto& w : _work)
    w /= float(f.height());
}

int MultiHostManager::chunks() const
{
  return _chunks;
}

void MultiHostManager::chunks(int n)
{
  _chunks = (_dist==EQUALLY || _dist==SPEED_PROPORTIONAL) ? DMP::csize() : n;
  if (_chunks==0) _chunks = DMP::csize();
}

void MultiHostManager::set_speeds()
{
  MPI_Status status;

  if (DMP::master())
  {
    _speed.resize(DMP::csize(),0.0);
    _speed[DMP::rank()] = _myspeed;
    for (int i=1; i<DMP::csize();i++)
    {
```

```
      float nspd;
      MPI_Recv(&nspd,1,MPI_FLOAT,MPI_ANY_SOURCE,NODESPD_TAG,MPI_COMM_WORLD,&status);
      _speed[status.MPI_SOURCE] = nspd;
    }

    _wspeed = 0.0;
    for (auto s : _speed)
      _wspeed += s;

    _rspeed.resize(DMP::csize());
    for (int r=0; r<int(_speed.size()); r++)
      _rspeed[r] = _speed[r]/_wspeed;
  }
  else
  {
    MPI_Send(&_myspeed,1,MPI_FLOAT,DMP::masterid(),NODESPD_TAG,MPI_COMM_WORLD);
  }
//  MPI_Bcast(&_wspeed,1,MPI_FLOAT,DMP::masterid(),MPI_COMM_WORLD);
}

void MultiHostManager::render(const Camera& c,FrameBuffer& f)
{
  Timer mtmr;

  mtmr.start();

  set_speeds();

  if (DMP::master())
  {
    mtmr.stop();
    logger::lock();
    logger::cinfo() << endl;
    logger::cinfo() << "World speed:  " << setw(3) << int(_wspeed) << endl;
    logger::cinfo() << "Node  speeds:" << endl;
    for (int r=0; r<int(_speed.size()); r++)
    {
      logger::cinfo() << "  node " << setw(3) << r
              << ": " << setw(3) << int(_speed[r])
              << " (" << setw(2) << int(100*_rspeed[r]) << "%)"<< endl;
    }
```

```
    logger::cinfo() << endl;

//    logger::cinfo() << "Number of processes: " << DMP::csize() << endl;

    logger::cinfo() << "World:    "
            << "[" << setw(5) << 0 << " -" << setw(5) << f.height()-1 << "]"
            << " " << setw(5) << f.height()
            << endl;
    logger::cinfo() << endl;

    logger::unlock();
    mtmr.start();
  }
#if 1
  mtmr.stop();
  MPI_Barrier(MPI_COMM_WORLD);
  mtmr.start();
#endif

  std::thread queueserver, receiver;

  if (DMP::master())
  {
    receiver = std::thread(&MultiHostManager::workReceiver,this,std::ref(f));
    if (_dist==WORK_QUEUE)
      queueserver = std::thread(&MultiHostManager::queueManager,this,f.height());
  }

  Timer tmr;

  tmr.start();

  vector<Duration> engrd(engines.size(),0);

  f.blank();

  bool more;
  while (split(f.height(),more))
  {
    logger::lock();
    logger::cinfo() << "Node " << setw(3) << DMP::rank() << ": "
```

```
            << "[" << setw(5) << myis << " -" << setw(5) << myie << "]"
            << " " << setw(5) << myie-myis+1
            << endl;
    logger::unlock();

#if 1
    vector<thread>  pool;

    int h = myie-myis+1;
    int w = f.width();
    int is = myis;
    int ie = myis-1;

    for (int i=0; i<int(engines.size()); i++)
    {
      is = ie + 1;
      ie = is + int(res[i]*(h-1));
      if (ie > (myis+h-1)) ie = myis+h-1;

      Job job(is,ie,0,w-1);
#if 0
      tmr.stop();
      logger::lock();
      logger::cinfo() << "Job for <" << engines[i]->name()
              << "> [node " << setw(2) << DMP::rank() << "]: "
              << job << " " << job.size() << endl;
      logger::unlock();
      tmr.start();
#endif
      pool.push_back(engines[i]->render(c,f,job));
    }

    for (auto& t : pool)
      t.join();

    for (int i=0; i<int(engines.size()); i++)
    {
      engrd[i] += engines[i]->rd;
    }

    join(f);
```

```
#endif
    if (!more)
      break;
  }

  tmr.stop();

  if (DMP::master())
  {
    if (_dist==WORK_QUEUE)
      queueserver.join();
    receiver.join();
  }
  mtmr.stop();

  stats(tmr.runtime(),mtmr.runtime(),engrd);

//  f.check();
}

void MultiHostManager::join(FrameBuffer& f)
{
  int buf[2];
  buf[0]= myis;
  buf[1]= myie;
  MPI_Send(&buf,2,MPI_INT,DMP::masterid(),CHNKTOJOIN_TAG,MPI_COMM_WORLD);
  if (DMP::slave())
  {
    int nr = myie-myis+1;
    int sz = nr*f.width()*SDF::nsamples;
    float* buffer = new float[sz];
    // row
    for (int i=0; i<nr; i++)
    {
      const vector<SDF>& row = f[myis+i];
      // col
      for (int j=0; j<f.width(); j++)
      {
        float* buf = buffer + (i*f.width()+j)*SDF::nsamples;
        for (int k=0; k<SDF::nsamples; k++)
          buf[k] = row[j][k];
```

```
      }
    }
    MPI_Send(buffer,sz,MPI_FLOAT,DMP::masterid(),DATAMATRIX_TAG,MPI_COMM_WORLD);
    delete[] buffer;
  }
}

void MultiHostManager::stats(const Duration& et, const Duration& met,
    const vector<Duration>& engrd) const
{
#if 1
  float  mx,mn,sd;
#if 0
  mx = 0.0;
  mn = 0.0;
  for (const auto& e : engines)
  {
    logger::cinfo() << "[perf]"
            << "[node " << setw(2) << DMP::rank() << "]"
            << "[" << e->name() << "]"
            << " " << e->rd.ms()
            << endl;
    if (e->rd.ms()>mx)
      mx = e->rd.ms();
    mn += e->rd.ms();
  }
  mn /= float(engines.size());
  sd = 0.0;
  for (const auto& e : engines)
  {
    float d = e->rd.ms() - mn;
    sd += d*d;
  }
  sd /= float(engines.size());
  sd = sqrt(sd);
#else
  mx = 0.0;
  mn = 0.0;
  for (int i=0; i<int(engines.size()); i++)
  {
    if (engrd[i].ms()>mx)
```

```
      mx = engrd[i].ms();
    mn += engrd[i].ms();
  }
  mn /= float(engines.size());
  sd = 0.0;
  for (const auto& d : engrd)
  {
    float dm = d.ms() - mn;
    sd += dm*dm;
  }
  sd /= float(engrd.size());
  sd = sqrt(sd);
#endif
  logger::lock();
  logger::cinfo() << "[perf]"
          << "[node " << setw(2) << DMP::rank() << "]"
          << "[all]"
          << " " << int(et.ms()) << endl;
  for (int i=0; i<int(engines.size()); i++)
  {
    logger::cinfo() << "[perf]"
            << "[node " << setw(2) << DMP::rank() << "]"
            << "[" << engines[i]->name() << "]"
            << " " << int(engrd[i].ms())
            << endl;
  }
  logger::cinfo() << "[perf]"
          << "[node " << setw(2) << DMP::rank() << "]"
          << "[bal]"
          << " " << fixed << setw(5) << setprecision(3) << mx/float(et.ms())
          << " " << fixed << setw(5) << setprecision(3) << mn/float(et.ms())
          << " " << fixed << setw(5) << setprecision(3) << sd/float(et.ms())
          << endl;
  logger::unlock();

#if 1
  MPI_Barrier(MPI_COMM_WORLD);
#endif

  if (DMP::master())
  {
```

```
vector<float> durations(DMP::csize(),0);
durations[DMP::rank()] = et.ms();
for (int i=1; i<DMP::csize();i++)
{
  MPI_Status status;
  float received;
  MPI_Recv(&received,1,MPI_FLOAT,MPI_ANY_SOURCE,DURATION_TAG,MPI_COMM_WORLD,&status);
  durations[status.MPI_SOURCE] = received;
}

mx = 0.0;
mn = 0.0;
for (int i=0; i<DMP::csize();i++)
{
  if (durations[i]>mx)
    mx = durations[i];
  mn += durations[i];
}
mn /= float(DMP::csize());
sd = 0.0;
for (auto nd : durations)
{
  float d = nd - mn;
  sd += d*d;
}
sd /= float(DMP::csize());
sd = sqrt(sd);

logger::lock();
logger::cinfo() << endl;
for (int i=0; i<DMP::csize(); i++)
{
  logger::cinfo() << "[perf]"
          << "[cluster]"
          << "[work]"
          << "[node " << setw(2) << i << "]"
          << " "
          << setw(2) << int(100*_rspeed[i])
          << " / "
          << setw(2) << int(100*_work[i])
          << endl;
```

```
    }
    logger::cinfo() << endl;
    logger::cinfo() << "[perf]"
            << "[cluster]"
            << "[all    ]"
            << " " << int(met.ms())
            << endl;
    for (int i=0; i<DMP::csize(); i++)
    {
      logger::cinfo() << "[perf]"
              << "[cluster]"
              << "[node " << setw(2) << i << "]"
              << " " << int(durations[i])
              << endl;
    }
    logger::cinfo() << "[perf]"
            << "[cluster]"
            << "[bal    ]"
            << " " << fixed << setw(5) << setprecision(3) << mx/float(met.ms())
            << " " << fixed << setw(5) << setprecision(3) << mn/float(met.ms())
            << " " << fixed << setw(5) << setprecision(3) << sd/float(met.ms())
            << endl;
    logger::unlock();
  }
  else
  {
    float nd = et.ms();
    MPI_Send(&nd,1,MPI_FLOAT,DMP::masterid(),DURATION_TAG,MPI_COMM_WORLD);
  }
#endif
}

}
}

#endif
```

2.2 Otros códigos

2.2.1 base/math.cc

```
#include "ftl/base/math.h"
```

```
#include <limits>
#include <cfloat>
#include <numeric>

namespace FTL {
namespace base {

const float eps    = 10*numeric_limits<float>::epsilon();
const float bigeps = 100*numeric_limits<float>::epsilon();
const float inf    = numeric_limits<float>::max();

vector<int> split(int n,int k)
{
  vector<int> r(k);

  int lk = n/k;
  int lx = n%k;
  for (int i=0; i<k; i++)
    r[i] = lk;
  for (int i=0; i<lx; i++)
    r[i]++;

//  int sum = std::accumulate(begin(r),end(r),0);

  return r;
}

vector<int> spliti(int n,int k)
{
  vector<int> r(k+1);

  vector<int> s = split(n,k);
  r[0]=0;
  partial_sum(begin(s),end(s),begin(r)+1);

  return r;
}

} // namespace base
} // namespace FTL
```