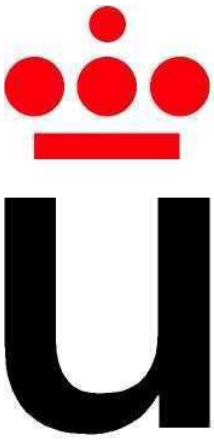

Universidad Rey Juan Carlos

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Curso Académico 2009 / 2010

Proyecto de Fin de Carrera

Paralelización del renderizado de animaciones en Pov-Ray

Autor: Elías Grande Rubio

Tutores: Óscar David Robles Sánchez
Pablo Toharia Rabasco

Agradecimientos

La realización de este proyecto no hubiera sido factible sin la ayuda y el apoyo recibido por parte de mis tutores y profesores, Óscar David Robles y Pablo Toharia, pertenecientes al Departamento de Arquitectura y Tecnología de Computadores y Ciencia de la Computación e Inteligencia Artificial de la Universidad Rey Juan Carlos, los cuales me han introducido en el mundo del cómputo en paralelo tanto en sistemas distribuidos así como en sistemas de memoria compartida. Por este motivo, es de agradecer, el tiempo que me han dedicado para resolverme todo tipo de dudas que me han ido surgiendo a lo largo del desarrollo de este trabajo, así como, las múltiples reuniones y revisiones a través de las cuales se me ha ido enfocando para elaborar esta importante tarea.

También tengo que agradecer el apoyo recibido por parte de José Luis Bosque, perteneciente al Departamento de Electrónica y Computadores de la Universidad de Cantabria, el cual me ha ayudado también desde tan lejos a que este proyecto sea posible.

Y en segundo lugar, y no por ello menos importante, quiero dar las gracias a mis padres y a mi hermana por apoyarme y animarme durante el desarrollo, no sólo de este proyecto, sino de toda la carrera.

Resumen

El proyecto detallado en esta memoria proporciona la capacidad de paralelizar el renderizado de animaciones tanto en sistemas distribuidos como en sistemas de memoria compartida. Con el fin de dar soporte a dicha capacidad, se ha desarrollado una aplicación informática combinando el lenguaje C y directivas MPI (del inglés *Message Passing Interface*) con *shell* de GNU/Linux, obteniendo de este modo, el comportamiento requerido para llevar a cabo la renderización en paralelo.

El sistema implementado sigue la estructura tradicional cliente-servidor, en la que se diferencian claramente los roles correspondientes al proceso servidor y los pertinentes de un proceso cliente. El proceso servidor actúa como un crupier repartiendo los paquetes de trabajo por hacer a los nodos clientes ociosos. Por su parte, los procesos clientes se encargarán de renderizar el paquete que se les ha asignado y devolverle después el trabajo realizado al proceso servidor.

La aplicación se ha llevado a cabo principalmente en lenguaje C utilizando la librería de MPI que se proporciona para dicho lenguaje, empleando para su desarrollo, programación estructurada. También cabe destacar que una buena parte del sistema ha sido realizado utilizando *shell* de GNU, puesto que este proyecto se ha llevado a cabo en su totalidad en sistemas GNU/Linux, que son uno de los ejemplos más prominentes de software libre.

Para concluir, merece la pena mencionar la utilidad del sistema implementado, el cual es capaz de distribuir el trabajo necesario para renderizar una animación entre el número de nodos disponibles que se desee, disminuyendo de este modo, con respecto al renderizado secuencial en un solo nodo, el tiempo de respuesta para obtener dicha animación renderizada, aprovechando la capacidad de cómputo de cada nodo lo máximo posible.

Índice

1. Introducción

Actualmente, una imagen generada mediante técnicas de render involucra una gran complejidad para su concepción, lo cual implica en sí mismo un elevado coste computacional. El algoritmo llevado a cabo en el renderizado de una imagen, independientemente de la complejidad de la escena descrita a renderizar, consta de etapas entre las que destacan, la transformación de vértices, ensamblado de primitivas, rasterización, texturado, coloreado, y operaciones finales como la eliminación de superficies ocultas [1]. La tendencia del ser humano a ambicionar imágenes y animaciones cada vez más realistas, provoca que los problemas a resolver sean cada vez más difíciles y complejos computacionalmente, lo cual implica la tendencia de los sistemas informáticos actuales a caracterizarse por una rápida evolución de los componentes del hardware que incrementan continuamente su capacidad.

Esta evolución es una propiedad deseable para todo sistema denominada escalabilidad. Una arquitectura escalable es aquella que tiene la capacidad de incrementar el rendimiento (idealmente de manera lineal) sin que tenga que rediseñarse y simplemente aprovechando el hardware adicional que se le apronte, mejorando de este modo la capacidad de servicio del sistema.

Entre las categorías existentes en las que se puede diferenciar el estilo arquitectónico de un sistema escalable [2], destacan como más importantes los modelos físicos que se distinguen por tener una memoria compartida común (Multiprocesadores de Memoria Compartida, del inglés *Shared-Memory Multiprocessors*) o una memoria distribuida no compartida (Multicomputadores de Memoria Distribuida, del inglés *Distributed-Memory Multicomputers*).

Existen varios modelos de multiprocesadores de memoria compartida: el modelo UMA (del inglés *uniform memory access*), el modelo NUMA (del inglés *nouniform memory access*), el modelo COMA (del inglés *cache-only memory architecture*) y el modelo cc-NUMA (del inglés *cache-coherente nouniform memory access*).

En el modelo UMA, la memoria física está uniformemente compartida por todos los procesadores y todos ellos tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede usar una caché privada y los recursos periféricos son compartidos de la misma forma que la memoria física.

El modelo NUMA, es un sistema de memoria compartida en el cual el tiempo de acceso varia dependiendo de la localización de las palabras en memoria. La memoria compartida esta físicamente distribuida para todos los procesadores, los cuales disponen de lo que se llama memoria local. El conjunto de todas las memorias locales forma un espacio de direcciones global accesible por todos los procesadores. Un procesador accede más rápidamente a su memoria local que a una memoria remota, lo cual provoca un retardo añadido producido por la red de interconexión.

Un caso especial de una máquina NUMA, es el basado en el modelo COMA, el cual establece que las memorias principales se convierten en cachés, eliminando de este modo la jerarquía de memoria de cada procesador, formando todas las cachés un espacio de direcciones global, y siendo atendido el acceso remoto a las cachés por la caché de directorios distribuidos.

El modelo cc-NUMA es el siguiente paso evolutivo, en el cual tanto la memoria caché como la memoria principal, son compartidas y direccionadas globalmente.

Un sistema de multicomputadores de memoria distribuida, es un sistema que consiste en múltiples computadores, a menudo llamados nodos, interconectados por una red de comunicaciones. Cada nodo es un computador autónomo compuesto de un procesador, memoria local y a veces discos duros o periféricos de entrada/salida. Todas las memorias locales son privadas y únicamente accesibles por los procesadores locales. Por esta razón, los tradicionales multicomputadores son conocidos como máquinas NORMA (memoria de acceso no remoto, del inglés *no-remote-memory-access*). También se conoce a este tipo de arquitectura compuesta por computadores independientes como *cluster*.

Con el fin de proporcionar una capa de abstracción sobre la complejidad y heterogeneidad de las redes subyacentes y facilitar la programación y el manejo de aplicaciones distribuidas sobre los sistemas mencionados con anterioridad, se utiliza un software cuya finalidad es interconectar aplicaciones para que puedan intercambiar datos entre ellas y crear así aplicaciones más grandes, denominado *middleware*. La figura 1.1 representa dichos conceptos.

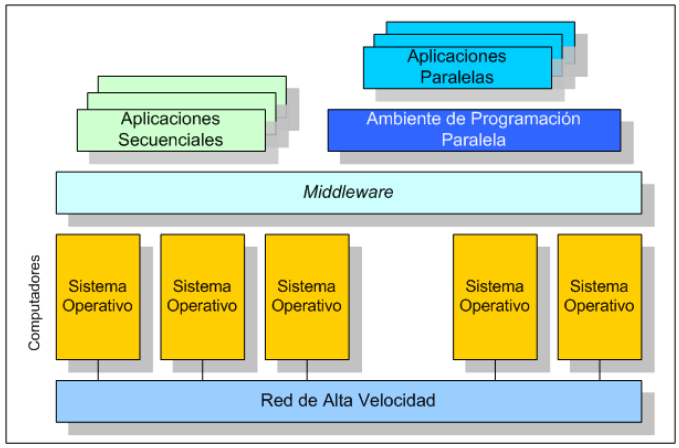

Figura 1.1 – Representación conceptual de los sistemas distribuidos y el uso de middleware

Un *middleware* proporciona un interfaz de programación de aplicaciones (API, del inglés *Application Programming Interface*) que sirve de capa de abstracción a otro software, el cual emplea las funciones proporcionadas por dicho API del *middleware*.

La API utilizada para la realización de este proyecto ha sido el interfaz de paso de mensajes (MPI, del inglés *Message Passing Interface*), el cual es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca diseñada para ser usada en programas que exploten la existencia de múltiples procesadores, permitiendo la sincronización entre procesos y la exclusión mutua.

Hay dos principios que caracterizan el paradigma de programación de paso de mensajes [3]. El primero es que se asume un espacio de direcciones repartido, y el segundo es que soporta sólo paralelización explícita.

En la realización de este proyecto se han utilizado diferentes implementaciones del estándar de MPI; *LAM/MPI Parallel Computing*, *MPICH: High-performance and Widely Portable MPI*, y *OPEN MPI: Open Source High Performance Computing*. Dichas plataformas son proyectos de código abierto que implementan el estándar MPI y proporcionan características como alto rendimiento soportando diversos sistemas operativos tanto de 32 como de 64bits, portabilidad, soporta gran variedad de redes y proporciona una única biblioteca que da soporte a distintos tipos de redes.

Como programa encargado de renderizar las animaciones, se ha utilizado la aplicación Pov-Ray. Pov-Ray (*Persistence of Vision Ray-tracer*) es un programa que utiliza la técnica de Ray-tracing consistente en la simulación una escena trazando los rayos de luz que el observador de la escena captaría desde su punto de vista. Estos rayos producen una escena realista, ya que van sufriendo todas aquellas condiciones impuestas en la escena, como sombras y reflejos.

Las imágenes se crean en ficheros de texto con la extensión ".pov", en un lenguaje descriptor de escena (en inglés *scene description language*), que sugiere cierta similitud con el lenguaje C. También nos permite la creación de animaciones moviendo objetos, cámaras y luces en función de un valor de reloj.

Al observar la siguiente imagen renderizada en la figura 1.2, se puede observar la calidad de imagen y el realismo que se pueden llegar conseguir con Pov-Ray.

Figura 1.2 – Imagen renderizada con Pov-Ray

2. Objetivos y metodología

2.1.Objetivos

El principal objetivo de este proyecto consiste en la realización de una aplicación que permita la renderización en paralelo de una animación en Pov-Ray. Se debe aprovechar al máximo el número total de nodos disponibles en el sistema, minimizando de este modo, el tiempo de respuesta hasta la obtención de la animación completamente renderizada. Mediante esta aplicación informática y la experimentación realizada que se aporte en esta memoria, el usuario podrá realizar su propio estudio en cuanto a la distribución de la carga de trabajo en la que dividir su animación, y el número óptimo de nodos necesarios para conseguir un nivel de eficiencia máximo entre los disponibles en el sistema.

Este objetivo principal se puede subdividir en una serie de subobjetivos que se enumeran a continuación:

- Realizar una aplicación que realice la renderización en paralelo de la animación independientemente de la arquitectura sobre la que se ejecute, ya sea un sistema de memoria compartida o un sistema de memoria distribuida.

- Implementar un algoritmo de equilibrio de carga que permita el máximo aprovechamiento de la capacidad de cómputo del sistema tanto para sistemas completamente dedicados como para sistemas no dedicados.

- Portabilidad entre las distintas plataformas implementadas del estándar de MPI.

- Realizar el correspondiente estudio experimental que ratifique el correcto funcionamiento de la aplicación realizada para las distintas arquitecturas ya mencionadas.

2.2.Descripción del problema

El realismo que se desea conseguir hoy en día a través de las imágenes y animaciones en 3D realizadas por ordenador es muy elevado. Para obtener dichas imágenes y animaciones se utilizan técnicas de render, las cuales se basan en realizar mediante computadora, un proceso de cálculo complejo por el que se interpreta una escena tridimensional descrita mediante una herramienta de diseño específica para ello, plasmando el resultado en una imagen bidimensional.

La gran capacidad de cómputo requerida para realizar dicho cálculo complejo en un tiempo de respuesta razonable es un factor muy importante a tener en cuenta en los computadores actuales. Las herramientas de diseño anteriormente citadas al igual que la gran mayoría de las aplicaciones contemporáneas que utilizamos habitualmente, realizan de manera secuencial todo el cálculo que requieren para ofrecernos los resultados que les solicitamos. Cuando el cálculo es muy complejo y costoso de realizar en términos computacionales, la realización de manera secuencial de dicho trabajo nos plantea la necesidad de buscar soluciones al tiempo de respuesta tan elevado que obtenemos.

Otro problema a tener en cuenta es la posibilidad de no disponer de un sistema dedicado exclusivamente a la realización de nuestro cómputo requerido, es decir, el nodo al que se le asigna el trabajo para realizarlo puede estar desarrollando simultáneamente otro trabajo distinto para otro usuario, o simplemente, para el usuario que solicita la renderización de su imagen o animación. Esta situación agravaría de manera considerable el tiempo que tarda dicho nodo en devolvernos el trabajo realizado si lo comparásemos con el tiempo que tardaría un sistema completamente dedicado.

Estos problemas plantean la disyuntiva de cómo dividir el trabajo que conlleva la realización de una imagen o animación, para intentar minimizar de algún modo, las consecuencias de los problemas aparejados con la ejecución secuencial.

La idea de compartir de algún modo el trabajo entre los posibles nodos existentes o simplemente de fragmentarlo con el fin de renderizar cada fragmento en el momento en el que el nodo del que disponemos esté más ocioso, nos hace enfocar el problema hacia cómo están estructurados los ficheros de nuestra herramienta específica que en este caso es Pov-Ray.

Una animación que se quiera renderizar mediante Pov-Ray, constará de dos ficheros inseparables; uno con extensión ".pov" que describe la escena mediante el lenguaje descriptor de escena proporcionado por dicha herramienta, y un fichero con extensión ".ini" que contiene datos de configuración sobre la animación, como número de fotogramas y variaciones de tiempos. A continuación se adjunta en la figura 2.1, un pequeño ejemplo de un fichero de extensión ".ini".

```
; Ejemplo de fichero INI
Input_File_Name=Prueba3.pov
Height=384
Width=512
Initial_Frame=1
Final_Frame=60
Initial_Clock=0.0
Final_Clock=2.0
Cyclic_Animation=on
```

Figura 2.1 – Ejemplo del contenido de un fichero de extensión ".ini" de Pov-Ray

A partir de la observación de la figura 2.1, se puede vislumbrar la opción de compartir el trabajo fragmentando la animación inicial en animaciones más pequeñas. Para conseguir dicho objetivo, la animación se dividiría en paquetes de un tamaño más pequeño, asignándole un rango de tiempo y un número de fotogramas menor a cada paquete. Pero esta idea, nos plantea una última cuestión consistente en cuál sería el tamaño de paquete óptimo y apropiado para dividir la animación, cuya respuesta no se puede obtener sin la correspondiente experimentación a partir de la cual se cotejen los resultados obtenidos y se planteen las conclusiones correspondientes.

2.3.Metodología empleada

La Ingeniería del Software se define [6] como, la disciplina de la ingeniería que realiza un uso apropiado de todas las teorías, métodos y herramientas para solucionar los problemas que aparecen, cuyo fin es encargarse de todos los aspectos relacionados con la producción de software, considerándose como tales, los procesos técnicos de desarrollo necesarios para producir y mantener software de calidad, desde sus etapas más tempranas de la especificación del sistema hasta el mantenimiento de dicho sistema tras su puesta en funcionamiento.

La producción del software se puede apoyar para su desarrollo en uno de los distintos modelos de desarrollo de los que dispone la Ingeniería del Software. Para la realización de este proyecto, se ha utilizado una metodología basada en el modelo incremental, el cual se creó como respuesta a las debilidades del modelo tradicional en cascada. En la figura 2.2 se muestra un esquema representativo del modelo de desarrollo citado.

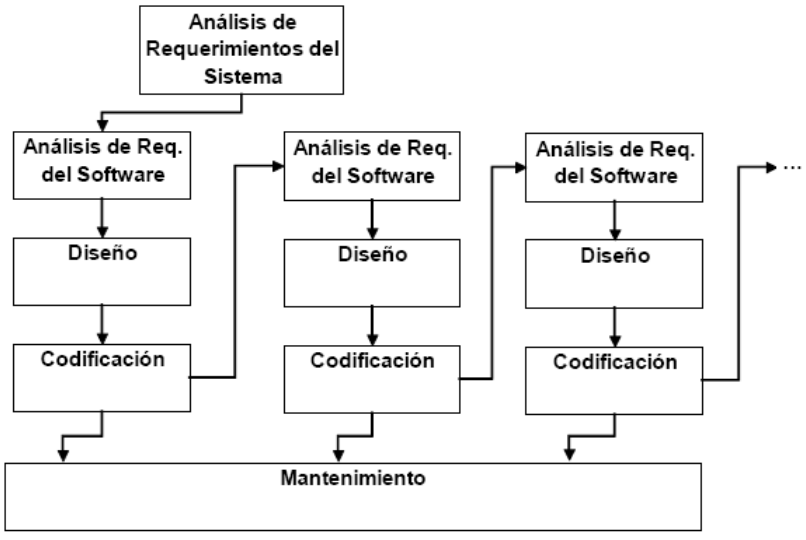

Figura 2.2 – Modelo incremental

El objetivo más importante del desarrollo incremental consiste en la capacidad de producir un programa de manera escalable permitiendo así al desarrollador aprender del incremento anteriormente realizado y proporcionando la capacidad de realizar cambios de diseño en el sistema para mejorarlo así como poder agregarle nuevas funcionalidades. Otra ventaja a tener en cuenta es lo probado que está cada incremento, ya que en las primeras versiones se va realizando una codificación simple, y en posteriores versiones se subsanan los errores que no se habían contemplado en la fase de pruebas y que aparecen una vez se ha puesto en marcha el sistema.

A continuación, se explica de manera concisa y breve las etapas que se han seguido del modelo de desarrollo iterativo e incremental para la realización de este proyecto.

Análisis de requerimientos del sistema

En esta etapa del desarrollo se valoran los principales objetivos y metas que se pretenden alcanzar y por los cuales se desea desarrollar la aplicación informática que dé soporte a dichas expectativas. Uno de los principales cometidos y más arduo conceptualmente, fue comenzar a pensar de manera concurrente, abstrayéndose de los sistemas distribuidos subyacentes y abandonando en gran medida, la forma a la que estamos acostumbrados hoy en día de realizar las tareas de manera secuencial.

Análisis de requerimientos del software

En esta fase se han valorado las metas de la etapa anterior para diagnosticar posibles riesgos o deficiencias en el desarrollo con el fin de que se puedan subsanar dichas deficiencias y abordar los riesgos latentes desde otra perspectiva para así eliminarlos o en su defecto minimizar sus consecuencias.

Diseño y codificación del software

En esta etapa del desarrollo de nuestra aplicación informática se lleva a cabo el diseño y la codificación del sistema con el fin de crear nuevas partes o mejorar nuestro proyecto mediante scripts y unidades que contienen los distintos tipos abstractos de datos necesarios, y corrigiendo en cada iteración del ciclo de desarrollo software todo tipo de error, ya sea correctivo (errores del producto), perfectivo (evolutivo, que cubre la expansión o cambios de necesidades del usuario), adaptativo (modificaciones por cambios en el entorno en el que el software opera) o preventivo (mejorar la calidad interna del sistema).

Mantenimiento del software

Al ser ésta la última etapa de cada iteración del ciclo de desarrollo software utilizado, se valoran los resultados obtenidos en esta iteración contrastándolos con los objetivos previamente marcados en las etapas iniciales de análisis y diseño. Si por algún motivo hubiera la necesidad de añadirle nueva funcionalidad a nuestro software, se iniciaría de nuevo una iteración más de nuestro ciclo de desarrollo para conseguir dar soporte a dichas funcionalidades. En caso contrario, si el software se considera completo, cumple todos los requerimientos y da soporte a toda la funcionalidad solicitada por el cliente, se da por finalizado el producto, comenzando de este modo, el mantenimiento posterior a la entrega del software.

3. Descripción informática

Como paso previo antes de comenzar con la explicación de la descripción informática que nos atañe, debería comprenderse qué es un *shell script* y las ventajas y desventajas aparejadas a la utilización del lenguaje C en la programación.

Un *shell script* es un archivo de texto, que contiene una serie de ordenes o mandatos para la *shell* (interprete de comandos del sistema), que dicho sistema interpreta de manera secuencial. La *shell* proporciona una interfaz de texto que sirve fundamentalmente para tres cosas: administrar el sistema operativo, lanzar aplicaciones y como entorno de programación.

Uno de los principales motivos que ha hecho que este proyecto sea desarrollado en lenguaje C es porque muchas librerías como la de MPI, están hechas para su uso en C. También cabe destacar del lenguaje C su eficiencia y su gran portabilidad debido a la gran variedad de compiladores existentes. Incluso sus características de bajo nivel no impiden que facilite la realización de programas modulares. Además hay que tener en cuenta que los sistemas GNU/Linux están desarrollados en este lenguaje, y por lo tanto, permite de manera fácil y flexible hacer uso de las llamadas al sistema.

En contrapartida, debido a la permisividad del compilador de C se llega a desarrollar código difícil de leer y de mantener lo cual conlleva una programación más lenta y cuidada que depende exclusivamente de la experiencia del programador. Un ejemplo claro de permisividad del compilador que más problemas suele dar a todo programador de lenguaje C es la gestión de memoria. La cantidad de riesgos que pueden surgir cuanto mayor es la duración al desarrollar y mantener cualquier aplicación en este lenguaje, conlleva que la gente conceptúe al lenguaje C, como una cuchilla de doble filo capaz de ayudarte o volverse contra ti.

3.1.Especificación

En esta parte de especificación de requisitos de nuestro proyecto software se indica lo que se espera que realice nuestro sistema una vez esté completamente implementado y operativo. Por este motivo, antes de comenzar a desarrollar nuestro software se vislumbran las funcionalidades básicas a las cuales debe dar soporte, haciendo una clara distinción entre los objetivos funcionales y los nos funcionales tenidos en cuenta a la hora de comenzar a realizar dicho proyecto.

Requisitos funcionales:

- El sistema software debe ser capaz de trocear una animación Pov-Ray en el número de paquetes que el usuario decida, representando cada paquete un trozo de la animación inicial distinto, de tal forma que si renderizamos todas las particiones obtenidas consigamos la misma animación que si renderizamos la animación inicial.
- El sistema software debe ser capaz de gestionar de manera transparente para el usuario el control del proceso maestro y de los procesos esclavos que se habiliten en cada ejecución, independientemente de la cantidad de esclavos que se disponga.
- El sistema software debe ser capaz de gestionar el reparto de los paquetes de trabajo entre los esclavos que disponga en cada ejecución de manera inteligente dependiendo de la capacidad de cómputo de cada uno de los nodos esclavos.
- El sistema software permitirá variar tanto el número de particiones obtenidas a partir de la animación inicial así como el número de nodos esclavos de los que se dispondrá para realizar el reparto de trabajo.
- El sistema software carecerá de interfaz gráfica y únicamente será invocado mediante línea de comandos.

Requisitos no funcionales:

- La parte del sistema que dé soporte a las funcionalidades de un proceso esclavo así como la parte que implemente las funcionalidades de un proceso maestro debe ser realizada en lenguaje C utilizando para ello cualquier implementación del estándar de MPI como *middleware*.
- El sistema debe dar la opción de poder ejecutarse mostrando la distribución de paquetes de trabajos asignados a cada nodo esclavo así como el tiempo de comunicaciones empleado entre los esclavos y el maestro además del tiempo total de ejecución del programa.
- El sistema debe ser robusto con el fin de no aparecer inestabilidades que comprometan la ejecución normal de la aplicación ya sea por la introducción del usuario de datos incorrectos así como cualquier anomalía que pueda devenir de una ejecución concurrente.
- El tiempo de respuesta debe ser óptimo con respecto al tamaño del problema (número de fotogramas de la animación y calidad de cada fotograma).
- El tiempo de comunicaciones empleado entre los procesos esclavos y el maestro debe ser el mínimo indispensable ya que es el causante de producir retardos innecesarios en el tiempo de respuesta de nuestro sistema software.
- El aprendizaje y manejo de la aplicación debe ser rápido y sencillo además de ser fácilmente entendible su funcionamiento interno de cara al usuario.

3.2.Estudio de alternativas

Para resolver y dar soporte al problema planteado anteriormente no existe una única solución concreta. Por este motivo, se puede plantear distintos diseños de nuestra aplicación que mediante la utilización de un paradigma de programación concurrente es capaz de resolver de manera eficiente el problema citado.

Mediante las siguientes alternativas que se explicarán a continuación se pretende alcanzar como resultado la obtención de una animación sobre un sistema distribuido de la manera más eficiente posible utilizando el mayor número de nodos disponibles de dicho sistema en cada momento, consiguiendo de este modo, un menor tiempo de respuesta en el renderizado de la animación.

Todas las alternativas que se plantean tienen en común la separación conceptualmente clara de los roles correspondientes a un proceso maestro y a un proceso esclavo. Dichos roles son, para el proceso maestro, en el caso de que queden paquetes de trabajo por renderizar, gestionar el correspondiente envío de paquetes de trabajo y la posterior gestión de los paquetes de imágenes renderizadas enviadas por los esclavos al proceso maestro. Por otra parte, el proceso esclavo recibiría el paquete de trabajo del proceso maestro, realizaría posteriormente el correspondiente renderizado de dicho paquete, y enviaría después el resultado obtenido al proceso maestro, haciéndole saber de este modo, que dicho proceso esclavo pasa a estar ocioso y esperando más paquetes de trabajo.

La forma de diferenciar una alternativa de otra es en cómo se realiza la gestión pertinente al envío de paquetes de trabajo con el fin de que los esclavos realicen el renderizado de la manera más eficientemente posible. A continuación, se explicará en detalle las distintas alternativas planteadas para la realización de la gestión del envío de los paquetes de trabajo.

Primera alternativa (Alternativa Crupier)

La primera alternativa que evaluamos se basa en un algoritmo en el cual el proceso maestro reparte inicialmente un paquete de trabajo a cada proceso esclavo, manteniéndose después a la espera de recibir las imágenes renderizadas. Cuando obtiene un paquete de imágenes renderizadas de un esclavo, en el caso en el que siga habiendo paquetes de trabajo por realizar, el proceso maestro le enviará al proceso esclavo que acaba de terminar, el siguiente paquete de trabajo por hacer. Por tanto, el proceso maestro actúa como un crupier, repartiendo paquetes del mismo tamaño a los esclavos ociosos mientras quede trabajo pendiente por hacer.

Segunda alternativa (Alternativa Adalid)

La segunda alternativa que valoramos consiste en un diseño algorítmico más sofisticado que el anterior, en el cual el proceso maestro reparte todos los paquetes a partes iguales entre los esclavos, y espera después a que éstos le devuelvan el trabajo realizado. Mientras aguarda la respuesta, el maestro comienza a preguntar a cada esclavo que porcentaje de paquetes lleva renderizado. Una vez que conoce el progreso de cada uno, le pide paquetes al esclavo más sobrecargado para enviárselo a otro esclavo que este con menos carga de trabajo con el fin de equilibrar la carga.

Tercera alternativa (Alternativa Piramidal)

La tercera alternativa consiste en realizar la fragmentación de la animación en paquetes de distintos tamaños, enviando al comienzo de la ejecución los paquetes más grandes, para continuar repartiendo los de tamaño un poco menor y así sucesivamente con el fin de ir equilibrando la carga de trabajo en función del trabajo pendiente por hacer sin la necesidad de estar preguntando a cada esclavo por su nivel de progreso. Esta alternativa es claramente un híbrido entre las dos anteriores.

3.3.Diseño

Para el desarrollo de este proyecto se ha utilizado el *middleware* de MPI facilitado por las implementaciones de Open MPI, LAM/MPI y MPICH, en lenguaje C. Al ser un estándar las bibliotecas tienen las mismas características, proporcionando una serie de funciones y tipos de datos muy útiles para la realización de un sistema concurrente basado en paso de mensajes.

A continuación se explican brevemente las funciones más importantes de la biblioteca utilizadas para el desarrollo de este proyecto [5].

- **int MPI_Init(int *argc, char ***argv)**

  Esta función es la encargada de iniciar la aplicación paralela. Independientemente del número de procesos disponibles (entre proceso maestro y esclavos), sólo se puede ejecutar una única vez por cada proceso y ha de ser la primera instrucción MPI que se ejecute.

- **int MPI_Comm_size(MPI_Comm comm, int *size)**

  Gracias a esta función, el proceso que la ejecuta es capaz de averiguar el número de procesos involucrados en la ejecución de la aplicación concurrente teniendo en cuenta tanto el proceso maestro así como los esclavos.

- **int MPI_Comm_rank(MPI_Comm comm, int *rank)**

  El proceso que ejecuta esta función es capaz de averiguar su identificador que lo reconoce dentro del conjunto de procesos que están involucrados en la ejecución de la aplicación.

- **int MPI_Finalize(void)**

  Esta función es la encargada de concluir con la ejecución de la aplicación. De igual forma que la función encargada de iniciar la aplicación, sólo se puede ejecutar una vez por cada proceso.

- **int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**

  Esta función es la encargada de difundir un mensaje desde el proceso con rango de "root" al resto de procesos involucrados en la ejecución.

- **int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)**

  Gracias a esta función, el proceso que la ejecuta se queda bloqueado esperando recibir un mensaje del emisor especificado en "source" y con la etiqueta definida en "tag".

- **int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)**

  Obtiene el número de elementos a "alto nivel" del mensaje recién recibido que aún no ha sido tratado con un "MPI_Recv".

- **int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position, MPI_Comm comm)**

  Gracias a esta función, se pueden empaquetar distintos tipos de datos dentro de un buffer con el fin de enviarlos en un único mensaje como un único paquete.

- **int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)**

  Se encarga de ir desempaquetando los datos contenidos en el paquete recibido empaquetados previamente por "MPI_Pack".

- **int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**

  Mediante esta función se envía un buffer de datos de un tipo determinado (aceptando buffer previamente empaquetados también) a un proceso con un identificador comprendido entre los posibles del conjunto de procesos en ejecución.

- **int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**

  Mediante esta función se recibe un buffer de datos de un tipo determinado (incluyendo buffer empaquetados) de un proceso con un identificador comprendido entre los posibles del conjunto de procesos en ejecución. Es una función bloqueante, lo cual quiere decir, que hasta que no haya un mensaje el proceso no seguirá ejecutándose.

Una vez detalladas las funciones más importantes del estándar de MPI así como sus propiedades intrínsecas, se muestra a continuación, las etapas más representativas del diseño en las que han sido empleadas.

Cabe mencionar del diseño general, que se ha optado por implementar la alternativa crupier comentada en el punto 3.2 de esta memoria. Se ha preferido escoger esta alternativa, ya que se considera que realiza un uso de la red eficiente, utilizando tamaños de paquetes iguales y enviando únicamente los mensajes imprescindibles.

3.3.1.Diseño sobre un sistema de memoria compartida

El sistema de memoria compartida sobre el que se ha realizado el primer diseño de nuestra aplicación ha sido sobre un sistema SGI PRISM que dispone de 32GB de memoria RAM y se ha configurado con dieciséis procesadores Intel Itanium 2, a 1500MHz. Intel Itanium es una arquitectura de 64 bits conocida como IA-64 (***I**ntel **A**rchitecture-**64***), basada en el modelo EPIC (del inglés *Explicitly Parallel Instruction Computing*).

Dicho sistema pertenece a la Escuela Técnica Superior de Ingeniería Informática de la Universidad Rey Juan Carlos y el nombre del *host* que identifica dicho sistema es “Nemea”.

El diseño de la aplicación que se debe realizar para que dé soporte a la especificación de requisitos mencionada con anterioridad en esta memoria, consiste en la fragmentación inicial de la animación a renderizar, para a continuación realizar la distribución del trabajo entre los distintos procesadores disponibles en el sistema. Se utilizará para ello, un *middleware* específico que nos abstraerá considerablemente de las conexiones y demás cálculos previos a la creación de las instancias de nuestro proceso maestro y sus respectivos procesos esclavos encargados del renderizado en paralelo.

El diagrama de actividad que muestra la figura 3.1 ilustra lo que debe hacer cada proceso y en qué orden para que el diseño de nuestra aplicación sea correcto y dé soporte a los requisitos exigidos.

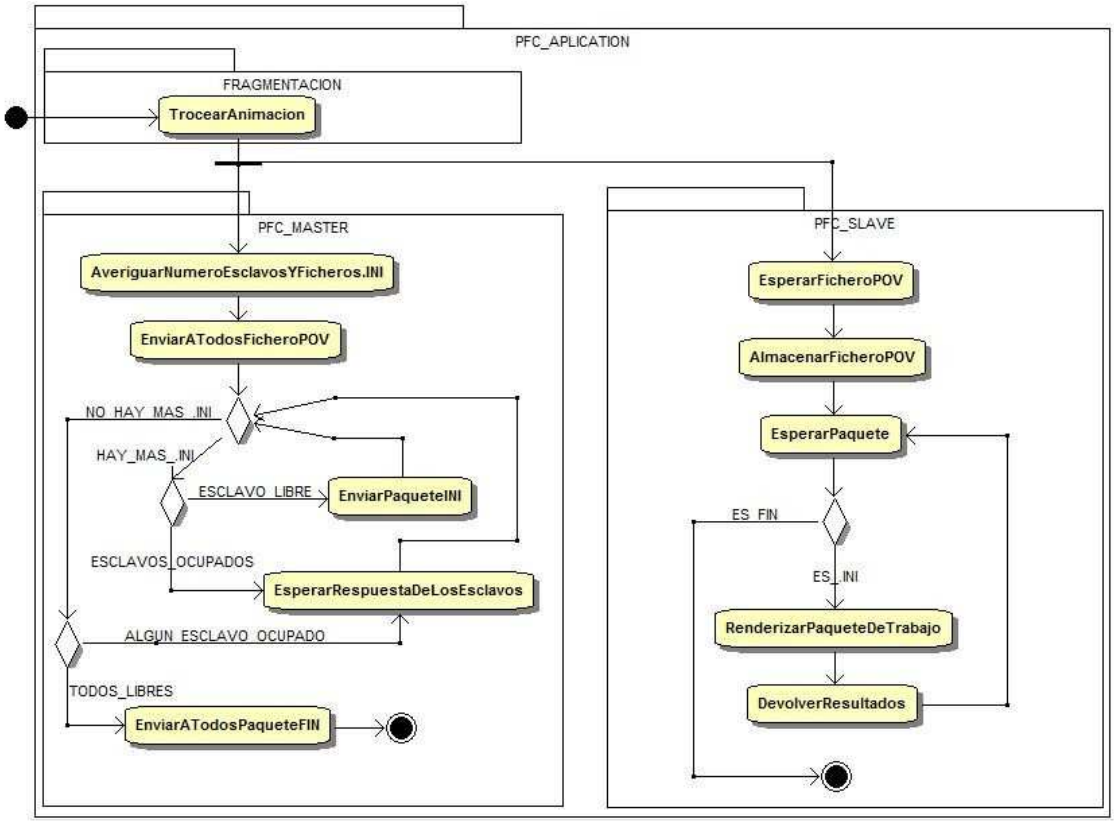

Figura 3.1 – Diagrama de actividad que representa el diseño de la aplicación para el sistema de memoria compartida

Como se puede observar en el diagrama de actividad de la figura 3.1, la aplicación se divide en tres módulos, cada uno de los cuales desempeña un rol específico dentro del algoritmo de nuestra aplicación.

El módulo encargado de la fragmentación de la animación realiza un algoritmo sencillo utilizando *shell* de GNU/Linux mediante el cual creará tantos ficheros de animación como divisiones se le especifiquen. El script realizará la comprobación de la validez de los parámetros así como la existencia del fichero de escena aparejado al fichero ".ini". Además, capturará los valores de reloj y fotogramas a partir de los cuales realizará los cálculos necesarios para la correcta fragmentación de la animación en paquetes de trabajo más pequeños. Con el fin de obtener siempre una distribución simétrica de los valores en los ficheros de animación creados incluso cuando el número de divisiones pedido no sea múltiplo del número de fotogramas, se ha diseñado el siguiente algoritmo en pseudocódigo:

```
Intervalo_reloj = abs(Final_Clock – Initial_Clock) / Num_divisiones
Valor_reloj = Initial_Clock
Intervalo_reloj_Fotograma = abs(Final_Clock – Initial_Clock) / (Final_Frame - Initial_Frame + 1)

if ((Final_Frame - Initial_Frame + 1) mod Num_divisiones = 0) then
        Intervalo_fotograma = (Final_Frame - Initial_Frame + 1) / Num_divisiones
        Valorframe = Initial_Frame – 1
        Aux = 1
else
        Intervalo_fotograma = (Final_Frame - Initial_Frame) / Num_divisiones
        Valorframe = Initial_Frame
        Aux = 0
for (1 to Num_Divisiones)
        if (Primera_Division) then
                Escribe_fichero_Initial_Frame(Valorframe + Aux)
                Escribe_fichero_Initial_Clock(Valor_reloj)
        else
                Escribe_fichero_Initial_Frame(Valorframe + 1)
                Escribe_fichero_Initial_Clock(Valor_reloj + Intervalo_reloj_Fotograma)

        if (Ultima_Division) then
                Escribe_fichero_Final_Frame(Final_Frame)
                Escribe_fichero_Final_Clock(Final_Clock)
        else
                Escribe_fichero_Final_Frame(Valorframe + Intervalo_fotograma)
                Escribe_fichero_Final_Clock(Valor_reloj + Intervalo_reloj)

        Valor_reloj = Valor_reloj + Intervalo_reloj
        Valorframe = Valorframe + Intervalo_fotograma
end for
```

Los otros dos módulos que aparecen en la figura 3.1, es decir, el módulo "PFC_MASTER" y el módulo "PFC_SLAVE", son los encargados de realizar el renderizado de la animación ya fragmentada de manera concurrente mediante la utilización de paso de mensajes. El algoritmo ilustrado en la figura 3.1 precisa los detalles necesarios para implementar la primera alternativa ya comentada en el apartado 3.2 de esta memoria. La primera alternativa era en la cual, mientras quedasen paquetes de trabajo pendientes por hacer, el proceso maestro actuaba como un crupier repartiendo los paquetes de trabajo del mismo tamaño a los esclavos ociosos.

La figura 3.2 que se muestra a continuación representa mediante un diagrama estático los submódulos en los que se ha dividido cada módulo ya mencionado y sus características principales.

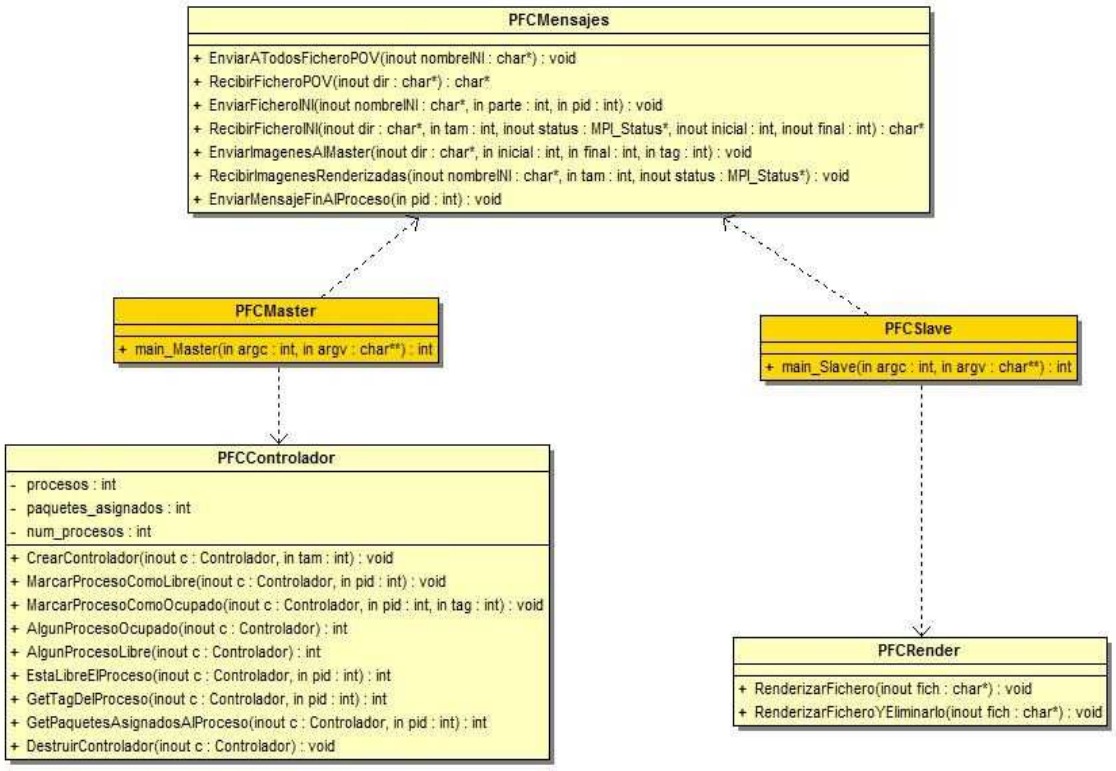

Figura 3.2 – Diagrama estático que representa el diseño de la parte concurrente de la aplicación para el sistema de memoria compartida

Como se puede observar en la figura 3.2, los módulos “PFCMaster” y “PFCSlave” son programas independientes con sus propias funciones “main” que utilizan los distintos submódulos representados en dicha figura. Este diseño ha sido condicionado por la implementación del *middleware* utilizado ya que permitía la opción de lanzar de manera sencilla dos programas independientes. Además, conceptualmente, a partir de este diseño es más sencillo entender que cada proceso es completamente independiente y que lo único que tiene en común un proceso esclavo y un proceso maestro es el correspondiente trasiego de mensajes.

Siguiendo con la figura 3.2, en ella se aprecian tres submódulos importantes para el correcto funcionamiento de los respectivos programas. A continuación, se explicará de manera breve las funciones que aportan cada uno de ellos y sus propiedades intrínsecas.

- **PFCMensajes:** Es el único módulo que tienen en común ambos programas. Como las funciones requeridas por el programa que implementa el rol del esclavo así como las requeridas por el que implementa el rol del maestro tienen bastantes cosas en común, se ha optado por un único módulo en vez de dos módulos para cada respectivo rol con el fin de obtener un código más fácil de mantener.
- **PFCControlador:** Es un módulo que representa un tipo abstracto de datos que el proceso maestro usa para controlar, valga la redundancia, la gestión de los procesos esclavos y el envío de paquetes a dichos procesos actuando de manera similar a un registro de datos.
- **PFCRender:** Es un módulo que únicamente utiliza el proceso esclavo y que se encarga de realizar la llamadas correspondientes desde el lenguaje C al sistema operativo para que realice el renderizado de los paquetes de trabajo en los que se ha troceado la animación.

3.3.2.Diseño sobre un sistema de memoria distribuida

El sistema de memoria distribuida sobre el que se ha realizado la modificación del diseño de nuestra aplicación para el sistema de memoria compartida ha sido sobre un sistema IBM, más concretamente, un cluster a gran escala que dispone de más de 200 computadoras. Cada una de dichas computadoras dispone de 4GB de memoria RAM y se ha configurado con dos procesadores PPC970FX altivec supported, a 2200MHz. Altivec es un estándar propiedad de Apple, IBM y Freescale Semiconductor; los cuales han diseñado un conjunto de instrucciones en coma fija y coma flotante que mejora el juego de instrucciones respecto al IA-32 de Intel.

Dicho sistema pertenece al Instituto de Física de Cantabria (Centro Mixto Consejo Superior de Investigaciones Científicas – Universidad de Cantabria) y el nombre del *host* que identifica dicho sistema es “Altamira”.

Como ya se ha comentado, se ha realizado una modificación del diseño de nuestra aplicación con respecto al diseño original desarrollado para nuestro sistema de memoria compartida. Dichas modificaciones han sido llevadas a cabo debido a la forma de ejecutar cualquier aplicación en un sistema de estas características. Dicho sistema no ejecuta la aplicación directamente según se le ordena por línea de comandos sino que dicha ejecución pasa a una cola de espera de la cual saldrá para ejecutarse en el momento en que estén disponibles el número de procesadores especificados para cada ejecución.

Por este motivo, el planteamiento inicial de fragmentar la animación en un solo procesador y a continuación proceder a realizar la distribución de trabajos entre un número de procesadores muy superior teniendo en cuenta que disponemos además de dos programas completamente independientes y distintos, se convierte en algo claramente inviable para la arquitectura del sistema de memoria distribuida del que disponemos en este momento.

Debido a esto se ha realizado un nuevo diseño que suple las deficiencias del primero para la arquitectura que nos atañe. Dicho diseño queda claramente definido en la figura 3.3 que se muestra a continuación.

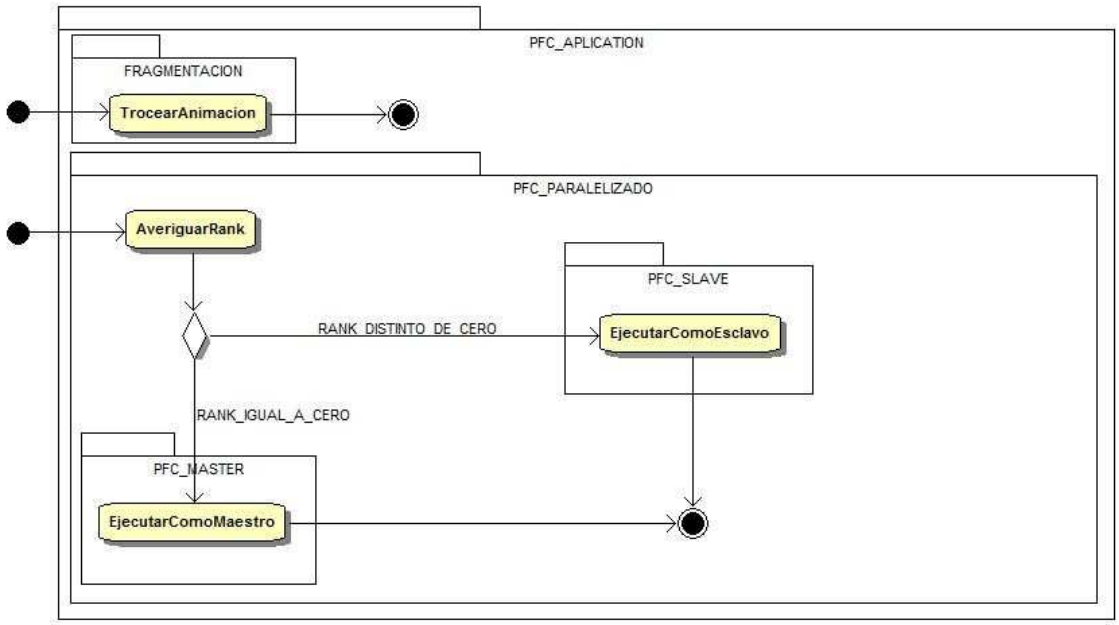

Figura 3.3 – Diagrama de actividad que representa el diseño de la aplicación para el sistema de memoria distribuida

Como se puede observar en la figura 3.3, se ha realizado un planteamiento en el cual primero se realiza una tarea que requerirá un procesador (fragmentar la animación), y una vez que esté completada se realizará la tarea de distribuir y renderizar dichos paquetes de manera concurrente entre el número de procesadores que se le indique en cada ejecución.

Debido a la restricción de poder ejecutar un único programa, se ha tenido que encapsular las aplicaciones del diseño anterior dentro de una misma aplicación. De este modo se realiza una única tarea en el número de procesadores que se requiera en cada momento y dependiendo del identificador de cada proceso dentro de la colección de procesos que están en ejecución, efectuará las operaciones como maestro o como esclavo según corresponda.

La figura 3.4 muestra mediante un diagrama estático la modificación explicada con anterioridad.

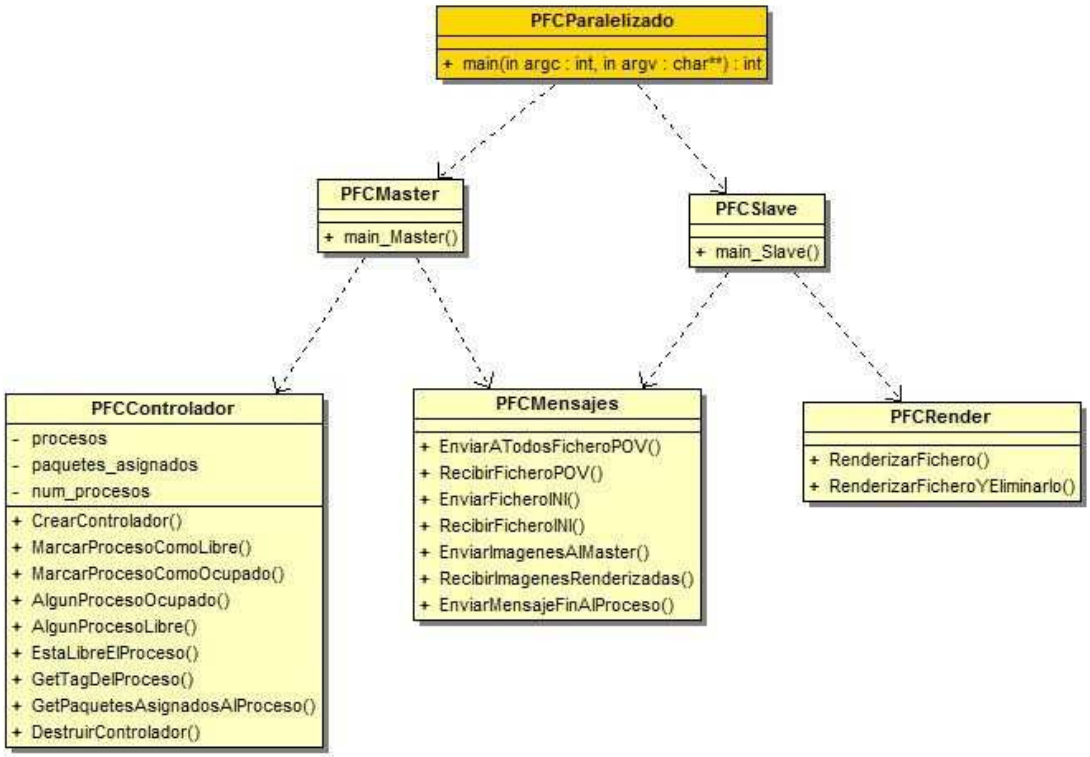

Figura 3.4 – Diagrama estático que representa el diseño de la parte concurrente de la aplicación para el sistema de memoria distribuida

Como se puede observar en la figura 3.4, se ha creado una jerarquía en la que el módulo "PFCParalelizado" es la nueva aplicación principal que engloba las aplicaciones principales del primer diseño para el sistema de memoria compartida. Esta nueva aplicación se encarga de usar según corresponda, las operaciones de un proceso maestro o un proceso esclavo dependiendo del identificador del que disponga dentro de la colección de procesos en ejecución.

3.4.Implementación

En esta sección de la memoria se detallará la codificación de los módulos, procedimientos y funciones que han sido implementados para llevar a cabo el desarrollo de esta aplicación.

Cabe mencionar que se ha seguido una metodología de implementación denominada "top-down", es decir, se ha comenzado implementado el algoritmo principal abstrayéndonos de los detalles, para posteriormente ir realizando refinamientos sucesivos con mayor número de detalles de implementación.

Módulo principal: PFCMaster

El algoritmo implementado en este módulo representa la sucesión de acciones que debe desarrollar un proceso maestro durante su ejecución para el correcto funcionamiento de la aplicación. Dicho algoritmo se puede dividir en tres partes fácilmente reconocibles las cuales son comentadas a continuación.

- Inicialización de la aplicación

```
int main(int argc,char* argv[]){
       //Inicializar Aplicación
       if(argc!=3){
              fprintf(stderr,"\nERROR: Núm. parámetros incorrectos.\n");
              exit(1);
       }
       int myrank,size;
       MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);
       //Enviar fichero .POV
       EnviarATodosFicheroPOV(argv[1]);
       //Fin Inicializar Aplicación
       ...
}
```

Como se puede observar en esta primera parte, primero se comprueba que el número de parámetros introducidos por línea de comandos es correcto, y posteriormente se procede a inicializar la aplicación con el fin de que el maestro conozca el número de esclavos de los que dispondrá. Para concluir, se realiza un envío a todos los esclavos disponibles del fichero de extensión ".pov" que será el que necesitarán todos los esclavos para poder renderizar la animación.

- Cuerpo del algoritmo del proceso maestro

```
int main(int argc,char* argv[]){
	...
	//Cuerpo PFCMaster
	Controlador c;
	CrearControlador(&c,size);
	int i=1;
	do{
		int j=1;
		while((i <= atoi(argv[2])) && AlgunProcesoLibre(&c)){
			if(EstaLibreElProceso(&c,j)){
				EnviarFicheroINI(argv[1],i,j);
				MarcarProcesoComoOcupado(&c,j,i);
				i++;
			}
			j++;
		}
		MPI_Status status;
		MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
			&status);
		int tam;
		MPI_Get_count(&status,MPI_CHAR,&tam);
		RecibirImagenesRenderizadas(argv[1],tam,&status);
		MarcarProcesoComoLibre(&c,status.MPI_SOURCE);
	} while(AlgunProcesoOcupado(&c) || (i <= atoi(argv[2])));
	//Cuerpo PFCMaster
	...
}
```

En esta segunda parte del algoritmo se aprecia con claridad la importancia del proceso maestro en el equilibrio de carga, el cual primeramente crea un controlador para ir registrando toda la información que requiere para una buena gestión de los esclavos. Posteriormente, mientras queden paquetes de trabajo pendientes por hacer, realiza una comprobación en su controlador en búsqueda de procesos esclavos ociosos a los que enviarles trabajo. Para finalizar, el proceso maestro se mantiene a la espera de respuesta de los esclavos y continúa enviando paquetes de trabajo a los que se queden ociosos hasta que todo el trabajo sea concluido con éxito.

- Finalización de la aplicación

```
int main(int argc,char* argv[]){
	...
	//Finalizar Aplicación
	int h;
	for(h=1;h<size;h++){
		EnviarMensajeFinAlProceso(h);	// TAG == 0
	}
	DestruirControlador(&c);
	MPI_Finalize();
	return 0;
}
```

Como se puede observar en esta última parte de finalización de la aplicación, el proceso maestro envía a todos los esclavos el mensaje de fin que les indica que no sigan esperando nuevos paquetes de trabajo ya que el renderizado de la animación ha concluido con éxito. Por este motivo, el esclavo que reciba dicho mensaje puede terminar su ejecución de manera normal y controlada.

Una vez que se ha terminado de enviar los mensajes de fin a todos los esclavos, se realizan las operaciones pertinentes a la liberación de memoria al carecer el lenguaje C de un recolector de basura como el que dispone Java o C#, finalizando posteriormente la aplicación concurrente.

Un detalle a tener en cuenta de la implementación llevada a cabo para el desarrollo de este módulo consiste en cómo a partir de la metodología de implementación “top-down” utilizada se ha implementado un algoritmo principal sin entrar en los detalles de implementación cumpliendo de este modo con el requisito no funcional de usabilidad, es decir, al desarrollar un algoritmo aislado de los detalles de la gestión interna de un proceso maestro y de un proceso esclavo, se podrá dar el mismo servicio aplicando algunos cambios sobre nuestro algoritmo inicial a otros tipos de plataformas de renderizado de imágenes.

Módulo principal: PFCSlave

El algoritmo desarrollado en este módulo principal representa qué debe hacer un proceso esclavo para el correcto funcionamiento de nuestro sistema concurrente. Dicho algoritmo se puede dividir de igual forma que el del proceso maestro en tres partes claramente distinguibles pero la más importante de ellas es aquella cuyo código se muestra a continuación.

```
int main(int argc,char* argv[]){
	//Inicializar Aplicación
	...
	//Recibir fichero .POV
	char* nombre = RecibirFicheroPOV("/tmp/");
	//Fin Inicializar Aplicación
	//Cuerpo PFCSlave
	for(;;){
		MPI_Status status;
		MPI_Probe(0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
		if(status.MPI_TAG==0){
			break;
		}
		else{
			int inicial,final,tam;
			MPI_Get_count(&status,MPI_CHAR,&tam);
			char* nombreINI = RecibirFicheroINI("/tmp/",
					tam,&status,&inicial,&final);
			RenderizarFicheroYEliminarlo(nombreINI);
			free(nombreINI);
			EnviarImagenesAlMaster("/tmp/",inicial,final,
					status.MPI_TAG);
		}
	}
	//Cuerpo PFCSlave
	//Finalizar Aplicación
	...
}
```

Como se puede observar en el código, la última instrucción que se ejecuta en la etapa de inicialización es para obtener el fichero de extensión ".pov" enviado por el proceso maestro a todos los esclavos.

El cuerpo del algoritmo consiste en mantenerse a la espera de mensajes por parte del proceso maestro. Si recibe un paquete de trabajo, lo renderiza y envía los resultados al maestro. En caso de que sea un mensaje de fin, el algoritmo realiza la finalización de la aplicación de manera normal y controlada.

Módulo secundario: PFCRender

La responsabilidad de renderizar los paquetes de trabajo que se le van asignando a cada esclavo recae sobre este módulo. Dicho módulo tiene el procedimiento necesario para realizar el renderizado del fichero que se le pase por parámetro. El código de dicho procedimiento se muestra a continuación.

```
void RenderizarFichero(char* fich){
       char* orden = GenerarOrden(fich,"povray -D 2>/dev/null ");
       int status = system(orden);
       free(orden);
       if(status){
              fprintf(stderr,"\nERROR al renderizar el fichero.\n");
              exit(status);
       }
}
```

De las instrucciones mediante las cuales el lenguaje C se puede comunicar con el sistema operativo, se ha optado por la instrucción a alto nivel "system()", evitando de este modo instrucciones de bajo nivel en las que incluir variables de entorno de la aplicación actual y demás parámetros.

Módulo secundario: PFCControlador

Este módulo es el encargado de facilitar el tipo abstracto de dato "controlador" al proceso maestro. Las operaciones que se permiten para utilizar este tipo de dato son las siguientes:

```
void CrearControlador(Controlador* c,int tam);
void MarcarProcesoComoLibre(Controlador* c,int pid);

// El valor de tag tiene que ser siempre mayor o igual a cero
void MarcarProcesoComoOcupado(Controlador* c,int pid,int tag);

int AlgunProcesoOcupado(Controlador* c);
int AlgunProcesoLibre(Controlador* c);
int EstaLibreElProceso(Controlador* c,int pid);
int GetTagDelProceso(Controlador* c,int pid);
int GetPaquetesAsignadosAlProceso(Controlador* c,int pid);
void DestruirControlador(Controlador* c);
```

Este módulo cumple los principios de abstracción, modularidad y encapsulación, facilitando una interfaz de operaciones para utilizar dicho tipo abstracto de dato, abstrayéndonos de este modo de la implementación interna del tipo.

Módulo secundario: PFCMensajes

En este módulo se aglutinan las funciones necesarias para comunicarse entre el proceso maestro y el esclavo. Se utiliza un único módulo para contener todas estas funciones con el fin de representar conceptualmente, el trasiego común de mensajes por parte de ambos procesos.

Dicho módulo facilita un interfaz con las siguientes funciones públicas, abstrayéndonos de cómo está implementado por debajo. Dicho interfaz se muestra a continuación.

```
//FUNCIONES PÚBLICAS
void EnviarATodosFicheroPOV(char* nombreINI);
char* RecibirFicheroPOV(char* dir);
void EnviarFicheroINI(char* nombreINI,int parte,int pid);
char* RecibirFicheroINI(char* dir,int tam,MPI_Status* status,
                        int* inicial,int* final);
void EnviarImagenesAlMaster(char* dir,int inicial,int final,int tag);
void RecibirImagenesRenderizadas(char* nombreINI,int tam,
                                 MPI_Status* status);
void EnviarMensajeFinAlProceso(int pid);
```

Como se puede observar en el conjunto de funciones públicas del interfaz de este módulo, se aportan exclusivamente las suficientes y necesarias requeridas por ambos procesos, de las cuales, el proceso maestro utiliza las que se encargan de enviar los mensajes de fin y los ficheros de extensión ".pov" y ".ini" así como la que se encarga de recibir por parte de cada esclavo los resultados de la renderización de los paquetes de trabajo, es decir, las imágenes renderizadas. Por otro lado, el proceso esclavo utiliza las funciones complementarias a las del maestro, es decir, las que permiten recibir los fichero ".pov" y ".ini" y la que permite enviar al maestro las imágenes renderizadas.

De las funciones anteriormente citadas, cabe destacar por su relevancia en la implementación de su código las siguientes:

- Función: *EnviarATodosFicheroPOV*

```
void EnviarATodosFicheroPOV(char* nombreINI){
       char* nombrePOV = CalcularNombrePOV(nombreINI);
       int tamContenido;
       char* contenido = CargarDatos(nombrePOV,&tamContenido);
       int tamNombre= strlen(nombrePOV);
       int outcount=(sizeof(MPI_INT)+tamNombre+tamContenido);
       char* buffer=(char*)malloc(sizeof(char)*outcount);
       int position=0;
       MPI_Pack(&tamNombre,1,MPI_INT,buffer,outcount,&position,
                MPI_COMM_WORLD);
       MPI_Pack(nombrePOV,tamNombre,MPI_CHAR,buffer,outcount,&position,
                MPI_COMM_WORLD);
       MPI_Pack(contenido,tamContenido,MPI_CHAR,buffer,outcount,
                &position,MPI_COMM_WORLD);
       MPI_Bcast(&outcount,1,MPI_INT,0,MPI_COMM_WORLD);
       MPI_Bcast(buffer,outcount,MPI_CHAR,0,MPI_COMM_WORLD);
       free(nombrePOV);
       free(contenido);
       free(buffer);
}
```

Como el fichero de extensión ".pov" va a ser utilizado por todos los procesos esclavos y además es el mismo para todos ellos, esta función se encarga de empaquetar en un mismo buffer, el tamaño del nombre del fichero, el nombre y el contenido de dicho fichero ".pov", para posteriormente enviarlo mediante "broadcast" a todos los esclavos.

Debido a la forma de ejecutarse la instrucción "MPI_Bcast", la cual se encarga de enviar mediante "broadcast" los mensajes de un proceso maestro a todos los esclavos, ha sido necesario antes de enviar el buffer con toda la información, realizar un envío previo. Dicho envío se encarga de hacer llegar a los esclavos la información sobre el tamaño que tendrá el buffer que recibirán a continuación con el fin de que puedan reservar la memoria necesaria para su correcto almacenamiento y posterior procesamiento.

- Función: *EnviarImagenesAlMaster*

```
void EnviarImagenesAlMaster(char* dir,int inicial,int final,int tag){
      char** contenidos = (char**)malloc(sizeof(char*)*
                                        (final-inicial+1));
      int* tamanyos = (int*)malloc(sizeof(int)*(final-inicial+1));
      int i,j;
      int tamEnviar= sizeof(MPI_INT)*2;
      for(i=inicial,j=0;i<=final;j++,i++){
            char* nombre = CalcularNombreImagenTemporal(dir,i);
            contenidos[j] = CargarDatos(nombre,&(tamanyos[j]));
            EliminarImagenTemporalCargada(nombre);
            free(nombre);
            tamEnviar += (sizeof(MPI_INT) + tamanyos[j]);
      }
      char* contenidoEnviar = (char*)malloc(tamEnviar*sizeof(char));
      int position = 0;
      MPI_Pack(&inicial,1,MPI_INT,contenidoEnviar,tamEnviar,
               &position,MPI_COMM_WORLD);
      MPI_Pack(&final,1,MPI_INT,contenidoEnviar,tamEnviar,
               &position,MPI_COMM_WORLD);
      for(i=inicial,j=0;i<=final;j++,i++){
            MPI_Pack(&(tamanyos[j]),1,MPI_INT,contenidoEnviar,
                     tamEnviar,&position,MPI_COMM_WORLD);
            MPI_Pack(contenidos[j],tamanyos[j],MPI_CHAR,
                     contenidoEnviar,tamEnviar,&position,
                     MPI_COMM_WORLD);
            free(contenidos[j]);
      }
      MPI_Send(contenidoEnviar,tamEnviar,MPI_CHAR,0,tag,
               MPI_COMM_WORLD);
      free(tamanyos);
      free(contenidos);
      free(contenidoEnviar);
}
```

Esta función es ejecutada por un proceso esclavo una vez que ha concluido de renderizar el paquete de trabajo que el maestro le ha asignado. Si observamos el código con atención, lo primero que realiza es la lectura de disco de todos los ficheros de imágenes para disponer de ellos en memoria. Posteriormente, empaqueta en un buffer el número del fotograma inicial y final del paquete con respecto a la animación inicial sin fragmentar y después va cargando los datos en el buffer siguiendo el patrón; primero el tamaño del contenido a cargar y luego el contenido cargado. Para finalizar, realiza el envío del mensaje al proceso maestro para liberar posteriormente la memoria principal utilizada durante su ejecución.

- Función: *RecibirImagenesRenderizadas*

```
void RecibirImagenesRenderizadas(char* nombreINI,int tam,
                                                    MPI_Status* status){
      char* buffer = (char*)malloc(sizeof(char)*tam);
      MPI_Recv(buffer,tam,MPI_CHAR,status->MPI_SOURCE,status->MPI_TAG,
               MPI_COMM_WORLD,status);
      int position=0;
      int inicial,final;
      MPI_Unpack(buffer,tam,&position,&inicial,1,MPI_INT,
                MPI_COMM_WORLD);
      MPI_Unpack(buffer,tam,&position,&final,1,MPI_INT,MPI_COMM_WORLD);
      int i,tamfich;
      char* contenido;
      for(i=inicial;i<=final;i++){
            MPI_Unpack(buffer,tam,&position,&tamfich,1,MPI_INT,
                      MPI_COMM_WORLD);
            contenido= (char*)malloc(sizeof(char)*(tamfich));
            MPI_Unpack(buffer,tam,&position,contenido,tamfich,MPI_CHAR,
                      MPI_COMM_WORLD);
            char* nombre = CalcularNombreImagenFin(nombreINI,i);
            EscribirImagen(nombre,contenido,tamfich);
            free(nombre);
      }
      free(buffer);
}
```

Una vez que el proceso maestro recibe un mensaje de un esclavo ejecuta esta función con el fin de procesarlo. Como se puede observar, lo primero que se realiza en el código es la recepción del mensaje. Posteriormente se procede a ir desempaquetando el mensaje recibido en el orden en el que ha sido empaquetado por el proceso esclavo. A la vez que se van desempaquetando las imágenes contenidas en el buffer recibido, se van escribiendo a disco dichas imágenes con el fin de ir liberando la memoria principal.

4. Experimentación y resultados

La parte con mayor importancia de este proyecto reside en la experimentación realizada y en los resultados obtenidos en los distintos sistemas distribuidos. En esta sección de la memoria se tratarán los distintos experimentos realizados en cada una de las diferentes arquitecturas.

Cabe mencionar previamente cuales son las medidas que se han calculado para realizar el estudio de la correspondiente experimentación. La medida más importante calculada es el "speedup", la cual representa cuántas veces es más rápida la ejecución del algoritmo paralelo que la correspondiente ejecución del algoritmo secuencial. Dicha medida se define mediante la siguiente formula, en la cual, p es el número de procesadores, T_1 es el tiempo de ejecución del algoritmo secuencial, y T_p es el tiempo de ejecución del algoritmo paralelo con p procesadores.

$$S_p = \frac{T_1}{T_p}$$

Se ha considerado el "speedup" como la medida más importante puesto que la otra medida de rendimiento, que es la eficiencia, no es más que dividir el correspondiente "speedup" entre el número de procesadores involucrados.

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

4.1.Experimentaciones preliminares

Debido a la incertidumbre provocada en las ejecuciones de una aplicación concurrente por posibles problemas derivados de la programación paralela, fue necesario testear el programa realizado antes de comenzar con las experimentaciones.

Por este motivo, la aplicación se testeó sobre un sistema de memoria compartida que dispone de 2GB de RAM y está configurado con dos procesadores Intel Core 2 DUO, a 1.66GHz. También se testeo sobre un sistema de memoria distribuida (cluster) de seis computadores, cada uno de los cuales dispone de 512MB de RAM y están configurados con un procesador AMD Athlon XP 1800+, a 1600MHz.

La implementación del *middleware* de MPI que ha dado soporte a las pruebas de experimentación preliminares de este proyecto en ambos sistemas ya comentados ha sido OPEN MPI.

Previamente se realizó una animación en Pov-Ray de un tamaño de 60 fotogramas que posteriormente serviría, tras realizar una serie de modificaciones sobre ella, como animación base en la realización de la experimentación posterior sobre los sistemas distribuidos a gran escala. A continuación, en la figura 4.1 se muestra uno de los fotogramas de dicha animación realizada también por el autor de este proyecto.

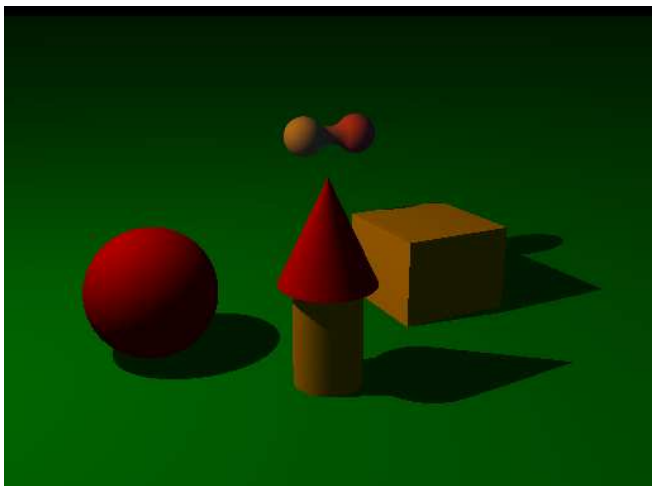

Figura 4.1 – Fotograma de la animación

4.2.Experimentación sobre un sistema de memoria compartida

Como ya se ha comentado previamente en esta memoria, la experimentación sobre un sistema de memoria compartida se ha realizado en "Nemea", computador que dispone de 32GB de RAM y dieciséis procesadores Intel Itanium 2, a 1500MHz.

Los experimentos se realizaron con la implementación del *middleware* de MPI de LAM/MPI que ya estaba instalado en la máquina. Se modificó la animación que se utilizó en las experimentaciones preliminares del sistema con el fin de que se convirtiera en una animación de 2000 fotogramas, la cual tarda en renderizarse de manera secuencial más de media hora (31 minutos y 9 segundos).

En dicho sistema se han realizado dos tipos de experimentos simulando distintas situaciones. Una de estas situaciones ha sido llevada a cabo con todos los procesadores desocupados representando de este modo un sistema dedicado. El otro tipo de experimentos ha sido llevado a cabo con un 25% de los procesadores ocupados con otros procesos que realizan gran cantidad de cómputo, para conseguir simular de esta manera un sistema no dedicado con el fin de estudiar el correcto equilibrio de carga implementado.

4.2.1.Simulación de un sistema dedicado

Los experimentos realizados han consistido en el reparto entre 2, 4, 8 y 16 esclavos de 2, 4, 8, 16, 32, 64, 128 y 256 particiones en las que se ha dividido la animación. Para cada combinación de esclavo con partición se ha realizado el experimento cinco veces con el fin de obtener un valor medio del tiempo de respuesta. Dichos tiempos (en segundos) se adjuntan en la tabla que se muestra a continuación.

		ESCLAVOS (Número de procesadores)			
Num. Particiones	Tam.Paquetes	2	4	8	16
2	1000	938.088			
4	500	939.913	473.037		
8	250	938.025	472.651	238.291	
16	125	938.772	472.517	238.124	122.638
32	63	946.656	481.769	250.199	136.663
64	32	947.063	482.547	250.172	139.624
128	16	978.695	529.560	304.460	199.274
256	8	1043.695	623.260	413.091	310.386

Tabla 4.1 – Tiempos de respuesta obtenidos en la experimentación con el sistema dedicado

A partir de los datos de la tabla 4.1 se ha obtenido su correspondiente representación gráfica, en la cual se observa que el tiempo de respuesta sigue una distribución negativa exponencial y está ligado al número y tamaño de las particiones que se reparten entre cada esclavo. Dicha representación se muestra en la figura 4.2.

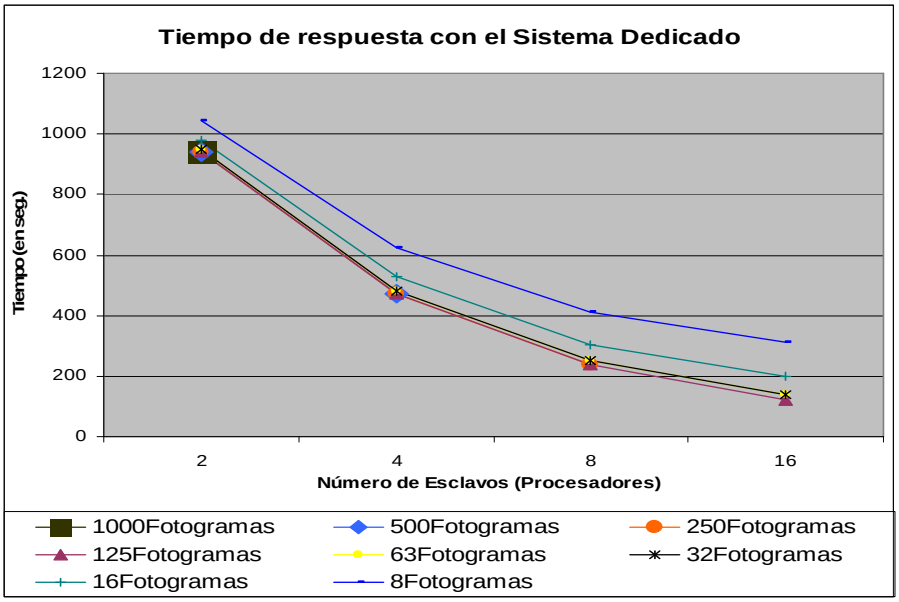

Figura 4.2 – Representación de los tiempos de respuesta obtenidos con el sistema dedicado

Con el fin de conocer cuántas veces es más rápida la ejecución del algoritmo paralelo respecto del secuencial, se calcula el "speedup" de cada resultado obtenido en los experimentos. Dicho "speedup" se muestra a continuación en la tabla 4.2.

		ESCLAVOS (Número de procesadores)			
Num. Particiones	Tam.Paquetes	2	4	8	16
2	1000	1.992			
4	500	1.988	3.950		
8	250	1.992	3.954	7.843	
16	125	1.990	3.955	7.848	15.239
32	63	1.974	3.879	7.469	13.675
64	32	1.973	3.873	7.470	13.385
128	16	1.909	3.529	6.138	9.378
256	8	1.790	2.998	4.524	6.021

Tabla 4.2 – Speedup sistema dedicado (Tiempo Secuencial / Tiempo con "P" procesadores)

Para poder clarificar los datos mostrados en la tabla anterior, se adjunta su correspondiente representación gráfica en la siguiente figura 4.3.

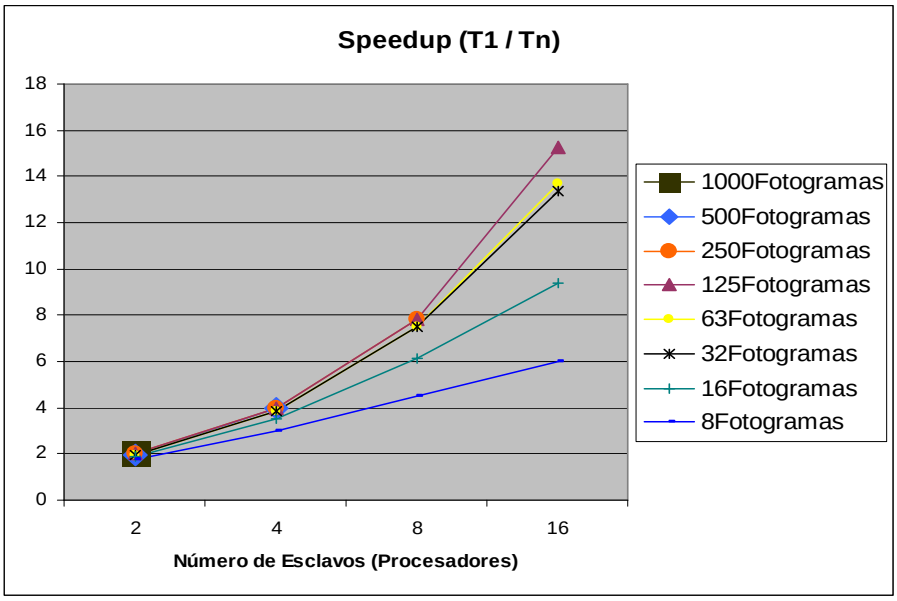

Figura 4.3 – Representación del speedup con el sistema dedicado

Podemos observar en la figura 4.3 que cuanto menor es el tamaño del paquete de trabajo, mayor es el número de paquetes a repartir y por lo tanto el "speedup" de nuestro sistema disminuye de manera considerable ya que el tiempo que se precisa para enviar los paquetes de tamaño más pequeño es mayor que el tiempo que se tarda en renderizarlos.

4.2.2.Simulación de un sistema no dedicado

Los experimentos realizados han consistido en el reparto entre 16 esclavos de 16, 32, 64, 128 y 256 particiones. Para simular un sistema no dedicado se ha ocupado un porcentaje de los procesadores con tareas extra con el fin de que no den respuesta rápida al renderizado de nuestra animación. Cada experimento se ha repetido cinco veces. En la tabla 4.3 se adjuntan los tiempos (en segundos) obtenidos.

		ESCLAVOS (Número de procesadores)				
Num. Particiones	Tam.Paquetes	16 (0Tarea)	16 (1Tarea)	16 (2Tareas)	16 (3Tareas)	16 (4Tareas)
16	125	122.638	240.552	360.031	480.322	600.948
32	63	136.663	191.623	192.212	240.580	302.086
64	32	139.624	163.711	182.416	241.610	171.440
128	16	199.274	212.228	220.525	223.717	229.492
256	8	310.386	329.206	336.729	338.528	342.293

Tabla 4.3 – Tiempos de respuesta obtenidos en la experimentación con el sistema no dedicado

A partir de los datos de la tabla 4.3 se observa que si el número de paquetes a repartir es igual al número de esclavos, el tiempo de respuesta está condicionado por los procesadores más ocupados, sin embargo, si el número de paquetes es mayor, los procesadores ocupados son contrarestados por los procesadores desocupados gracias al algoritmo de equilibrio de carga implementado. La figura 4.4 muestra esta realidad.

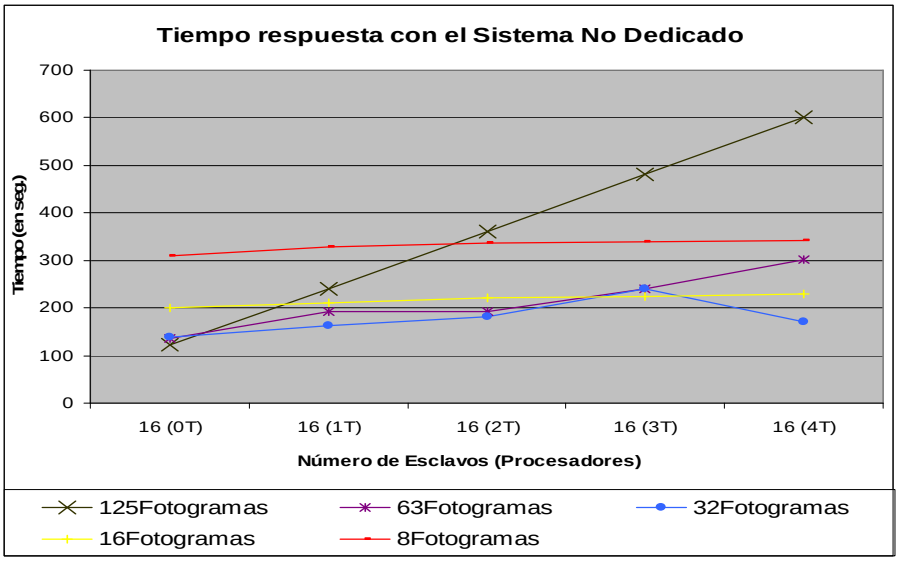

Figura 4.4 – Representación de los tiempos de respuesta obtenidos con el sistema no dedicado

Al ser nuestro sistema simulado un sistema no dedicado, el "speedup" dependerá de los procesadores más ocupados, por lo que cuantas más tareas extras tengan dichos procesadores, menor será el "speedup" obtenido. La siguiente tabla muestra el "speedup" calculado a partir de los resultados de la experimentación obtenidos.

		ESCLAVOS (Número de procesadores)				
Num. Particiones	Tam.Paquetes	16 (0T)	16 (1T)	16 (2T)	16 (3T)	16 (4T)
16	125	15.239	7.769	5.191	3.890	3.109
32	63	13.675	9.753	9.723	7.768	6.186
64	32	13.385	11.416	10.245	7.735	10.901
128	16	9.378	8.806	8.474	8.353	8.143
256	8	6.021	5.677	5.550	5.520	5.460

Tabla 4.4 – Speedup sistema no dedicado
(Tiempo Secuencial / Tiempo con "P" procesadores)

Como se puede observar en los datos de la tabla 4.4, el reparto equitativo de trabajo en un sistema no dedicado provoca una degradación excesiva del "speedup". Sin embargo, realizando un equilibrio de carga a partir de un mayor número de paquetes de trabajo más pequeños, mitiga en gran medida la degradación excesiva respecto de un sistema completamente dedicado. La siguiente figura muestra gráficamente los datos obtenidos en la tabla 4.4.

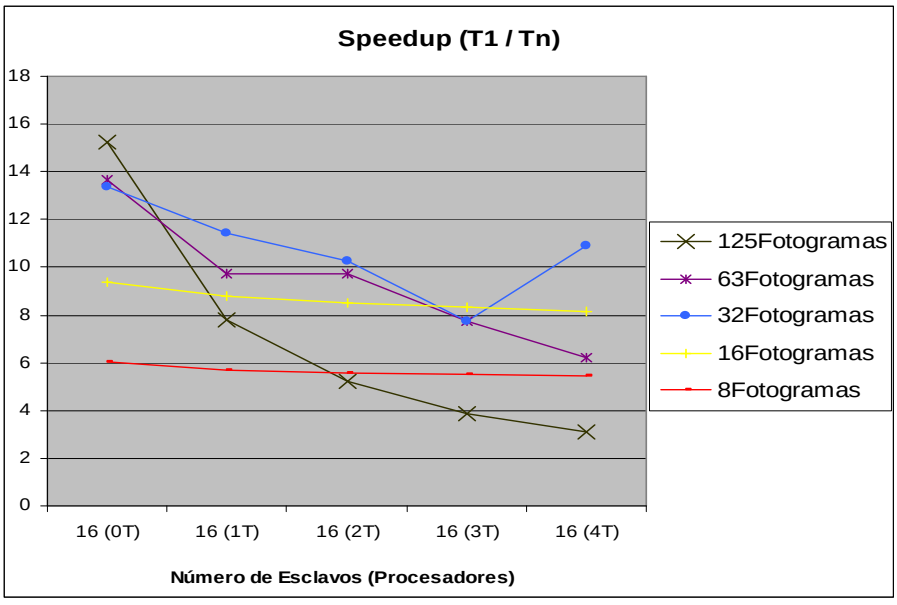

Figura 4.5 – Representación del speedup con el sistema no dedicado

La figura 4.6 muestra como se realiza el equilibrio de carga en los sistemas no dedicados simulados, representando para cada procesador el número de paquetes recibidos durante la ejecución. Los procesadores con identificadores 2, 3, 4 y 5 simulan los nodos con baja capacidad de cómputo al haberles asignados 1, 2, 3 o 4 tareas extra a parte de la de renderización. El reparto equitativo en un sistema dedicado queda representado cuando no se le asigna ninguna tarea extra a ningún nodo (0T).

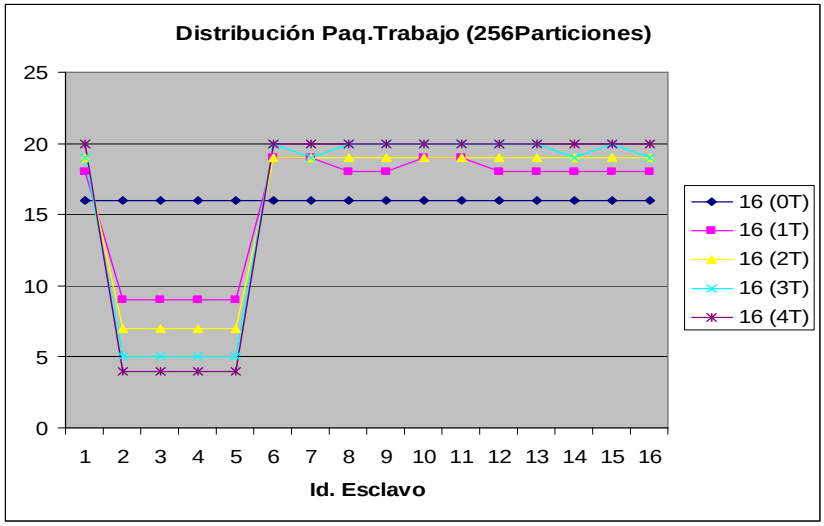

Figura 4.6 – Equilibrio de carga realizado

También cabe destacar como resultado relevante que cuanto mayor es el número de particiones a repartir, mayor es el porcentaje del tiempo de respuesta que se emplea en comunicación. La figura 4.7 representa claramente esta idea.

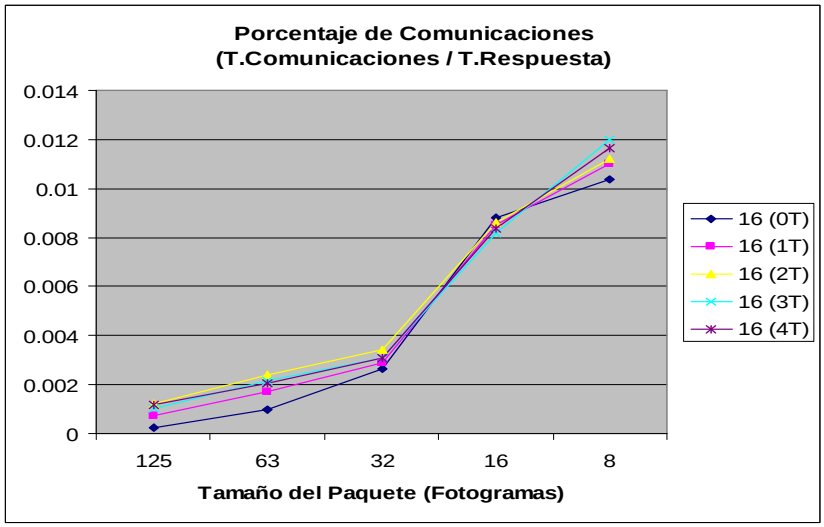

Figura 4.7 – Representación del porcentaje de comunicaciones

4.3.Experimentación sobre un sistema de memoria distribuida

Como ya se ha comentado en esta memoria, la experimentación sobre un sistema de memoria distribuida se ha realizado en "Altamira", un cluster a gran escala de más de 200 computadoras cada una de las cuales dispone de 4GB de memoria RAM y se ha configurado con dos procesadores PPC970FX altivec supported, a 2200MHz.

Los experimentos se realizaron con la implementación del *middleware* de MPI de MPICH que ya estaba instalado en la máquina. Se aprovechó la animación de 2000 fotogramas que se utilizó en las experimentaciones en el sistema de memoria compartida, la cual tarda en renderizarse de manera secuencial en este sistema más de media hora (33 minutos y 35 segundos).

Debido a la forma que tienen de ejecutarse las aplicaciones en este sistema de memoria distribuida, no se ha podido realizar los experimentos sobrecargando los nodos con tareas extras, por lo tanto únicamente se han realizado los experimentos con el sistema completamente dedicado.

Los experimentos realizados han consistido en el reparto de un número de paquetes de trabajo igual al número de esclavos disponibles en cada ejecución. El número de esclavos disponibles ha variado entre 2, 4, 8, 16, 32, 64, 128 y 256 esclavos. Para cada combinación de esclavo con partición se ha realizado el experimento treinta veces con el fin de obtener un valor medio del tiempo de respuesta lo más aceptable posible. Dichos tiempos (en segundos) se adjuntan en la tabla que se muestra a continuación.

	ESCLAVOS (Número de procesadores)							
Número Particiones	2	4	8	16	32	64	128	256
Igual al nº Esclavos	1406.330	881.556	714.585	442.118	307.767	191.201	131.569	126.121

Tabla 4.5 – Tiempos de respuesta obtenidos en el sistema de memoria distribuida

A partir de los datos de la tabla 4.5 podemos observar que se produce un decremento del tiempo de respuesta según vamos incorporando procesadores a nuestro sistema. La correspondiente representación gráfica de dichos datos se muestra reflejada en la figura 4.8.

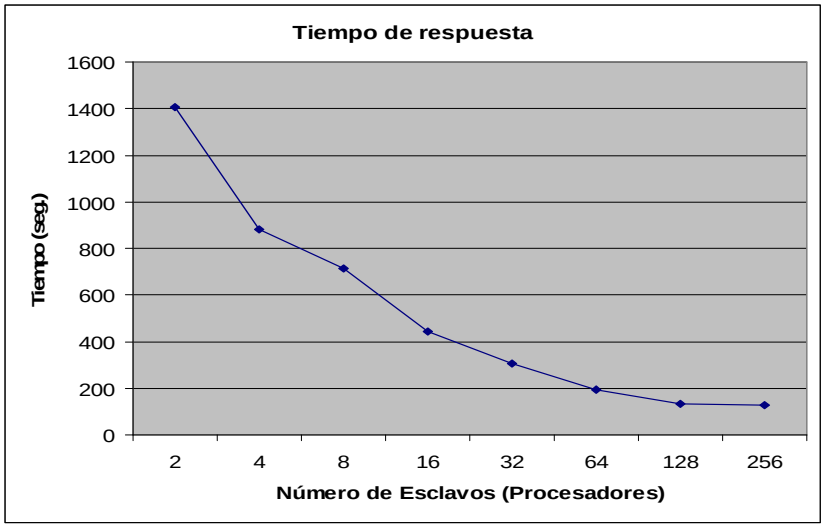

Figura 4.8 – Representación de los tiempos de respuesta obtenidos en el sistema de memoria distribuida

Una vez disponemos de los tiempos de respuesta, se calcula el "speedup" de cada resultado obtenido en los experimentos con el fin de conocer cuántas veces es más rápida la ejecución paralela que la ejecución secuencial. Se puede comprobar entonces, que el "speedup" de nuestro sistema crece de manera muy tenue según se le van incorporando más procesadores a la experimentación que actúen como procesos esclavos. La siguiente tabla 4.6 muestra el "speedup" calculado de cada uno de los experimentos ya mencionados.

	ESCLAVOS (Número de procesadores)							
Número Particiones	2	4	8	16	32	64	128	256
Igual al nº Esclavos	1.432	2.285	2.819	4.557	6.546	10.538	15.314	15.976

Tabla 4.6 – Speedup sistema de memoria distribuida (Tiempo Secuencial / Tiempo con "P" procesadores)

Con el fin de entender mejor los datos mostrados en la tabla 4.6, a continuación se añade su correspondiente representación gráfica en la figura 4.9 en la cual se observa claramente este leve crecimiento del "speedup".

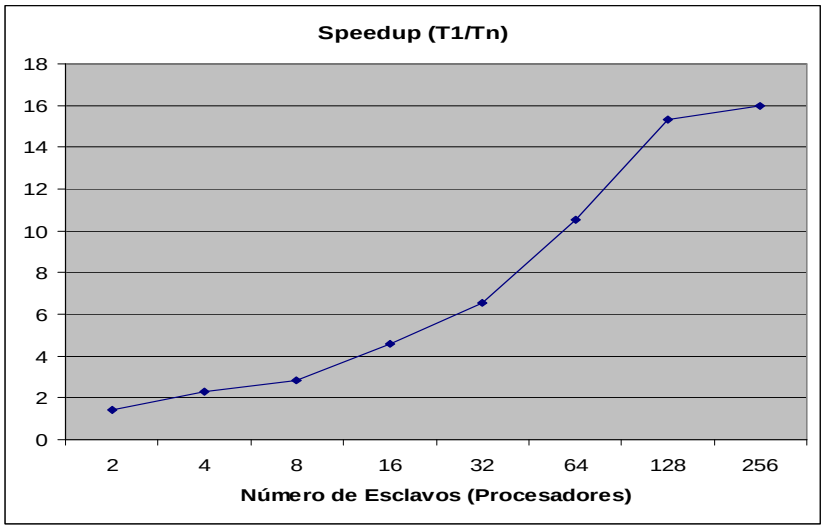

Figura 4.9 – Representación del speedup en el sistema de memoria distribuida

Este sutil crecimiento observado en la figura 4.9 no debe considerarse como algo completamente positivo para nuestra experimentación. Nuestro sistema está disminuyendo su tiempo de respuesta pero no de la manera que teóricamente se esperaría, es decir, se esta produciendo una degradación en la eficiencia de nuestro sistema según vamos incorporando procesadores. La figura 4.10 muestra la evolución de la eficiencia de nuestro sistema según incorporamos más esclavos en nuestra ejecución.

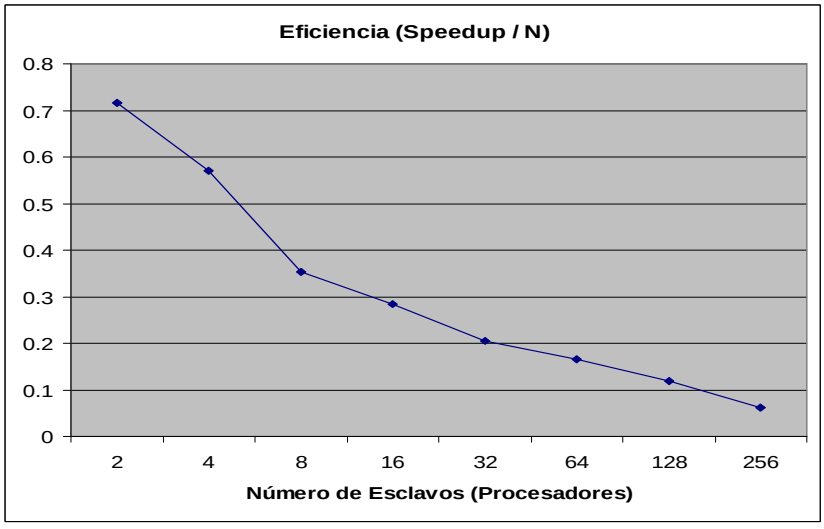

Figura 4.10 – Representación de la evolución de la eficiencia en el sistema de memoria distribuida

Tras realizar un estudio en profundidad de las posibles razones por las cuales se podía producir la degradación de la eficiencia en los experimentos, se ha conseguido acotar el problema de manera sencilla. El tamaño de problema utilizado para un sistema de estas características es demasiado reducido para obtener un sistema escalable, es decir, el tamaño de problema actual de 2000 fotogramas de nuestra animación es demasiado pequeño y nuestro sistema emplea más tiempo en comunicación entre procesos según va aumentando el número de procesadores, que en el renderizado de la animación. La siguiente figura 4.11 muestra de manera gráfica el alto porcentaje de tiempo de comunicación empleado en los experimentos realizados.

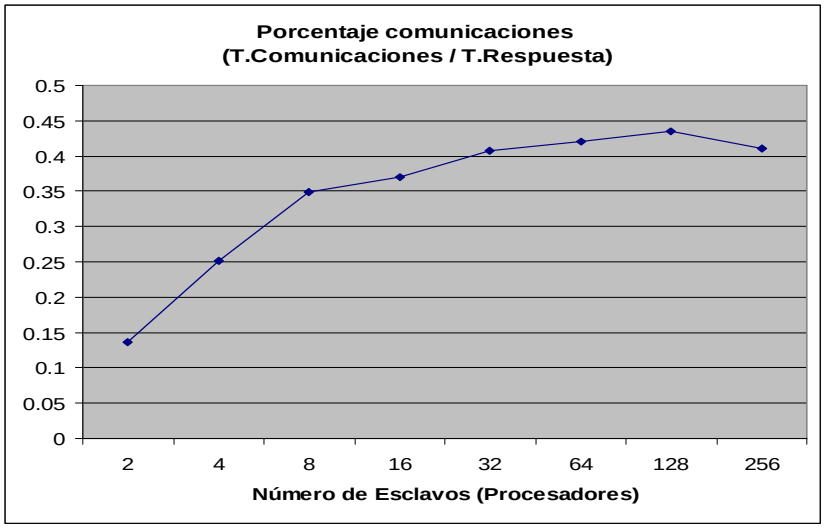

Figura 4.11 – Representación del porcentaje de comunicaciones en el sistema de memoria distribuida

Por este motivo, con el fin de evitar al máximo la degradación de la eficiencia de nuestro sistema cuando va aumentando el número de procesadores que actúan como procesos esclavos, existirían dos formas de aumentar el tamaño del problema a resolver para obtener un mayor porcentaje de tiempo de renderizado de los fotogramas con respecto al porcentaje de tiempo de comunicación entre procesos. Una de estas formas sería llevar a cabo una animación cuyos fotogramas requieran mayor capacidad de cómputo que el de la animación estudiada en esta memoria. Otra forma, sería la de incrementar de manera considerable el número de fotogramas de la animación manteniendo la actual capacidad de cómputo requerida para renderizar un fotograma.

A partir de los datos obtenidos en la experimentación realizada en este sistema de memoria distribuida y utilizando para aumentar el tamaño del problema la solución de incrementar de manera considerable el número de fotogramas de nuestra animación, se ha decidido realizar experimentos que confirman nuestra hipótesis de partida de que el tamaño de problema utilizado inicialmente para nuestro sistema era demasiado pequeño. Los experimentos realizados han consistido en la ejecución del renderizado de animaciones de 20000 y 40000 fotogramas, repitiendo cada experimento veinte veces para obtener un valor medio del tiempo de respuesta lo más aceptable posible. La siguiente tabla 4.7 muestra los resultados obtenidos con tamaños de problema mayor repartido entre 256 procesadores que actúen como procesos esclavos.

Fotogramas	Tiempo Secuencial	Tiempo Paralelo	Speedup
2000	2014.925	126.120	15.976
20000	20149.257	168.769	119.389
40000	40298.514	324.464	124.200

Tabla 4.7 – Comparación de los speedup para tamaños de problema mayores y el speedup del experimento inicial

Como se puede observar en la tabla 4.7, cuanto mayor es el tamaño del problema, el "speedup" de nuestro sistema aumenta de la manera que teóricamente esperábamos.

5. Conclusiones y trabajos futuros

A partir de las pruebas realizadas y una vez analizados los resultados obtenidos de dicha experimentación, se puede concluir esta memoria afirmando que se ha logrado satisfacer los objetivos propuestos al comienzo de este proyecto. Dicho proyecto deja abiertas líneas de trabajo para continuar con su desarrollo y su correspondiente estudio en el futuro.

5.1.Logros principales obtenidos y conclusiones

Se ha desarrollado una aplicación cuya principal finalidad consiste en permitir la renderización en paralelo de una animación en Pov-Ray. Dicha aplicación, aprovecha al máximo el número total de nodos disponibles en el sistema, minimizando de este modo, el tiempo de respuesta hasta la obtención de la animación completamente renderizada independientemente de la arquitectura sobre la que se ejecute, ya sea un sistema de memoria compartida o un sistema de memoria distribuida.

Se ha implementado un algoritmo de equilibrio de carga que permite un mayor aprovechamiento de la capacidad de cómputo del sistema mejorando considerablemente la eficiencia, tanto para sistemas completamente dedicados como para sistemas no dedicados.

Finalmente, se ha realizado una amplia experimentación con el fin de observar el correcto comportamiento de esta aplicación en arquitecturas de memoria compartida así como de memoria distribuida, simulando también sistemas dedicados y sistemas no dedicados. Se ha utilizado para todos los experimentos una variación considerable del número de procesadores, los cuales han variado desde dos procesadores hasta doscientos cincuenta y seis procesadores, con el fin de observar el funcionamiento apropiado tanto para sistemas pequeños como para sistemas a gran escala.

5.2.Posibles trabajos futuros

Como ya se ha comentado, este proyecto deja abiertas líneas de trabajo para continuar con su desarrollo y su correspondiente estudio en el futuro, de las cuales se sugieren como posibles las que se comentan a continuación.

- Implementación de las distintas alternativas planteadas para el algoritmo de equilibrio de carga con el correspondiente estudio de cada una ellas, así como el estudio comparativo de las tres alternativas.

- Incluir el algoritmo de fragmentación de la animación dentro del algoritmo del proceso maestro para su correspondiente estudio comparativo con la versión del módulo de fragmentación de la animación independiente.

- Utilización del algoritmo implementado en este proyecto para dar el mismo servicio a otras herramientas específicas de renderizado de imágenes.

6. Bibliografía

[1] "3D Computer Graphics". Alan Watt. Pearson/Addison Wesley, (2000).

[2] "Advanced Computer Architecture: Parallelism, Scalability, Programmability". Kai Hwang. McGraw-Hill Series in Computer Science, (1993).

[3] "Introduction to Parallel Computing". Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. Addison Wesley, (2003).

[4] "Parallel Programing". Barry Wilkinson, Michael Allen. Prentice Hall, (1999).

[5] "Parallel Programming with MPI". Peter Pacheco. Morgan Kaufmann, (1997).

[6] "Ingeniería del Software: un enfoque práctico". A.R.S. Pressman. McGraw-Hill, (1996).

[7] "Unix y Linux: guía práctica". Sebastián Sánchez Prieto, Óscar García Población. Ra-Ma, (2004).

[8] "ANSI C a su alcance". Herbert Schildt. Osborne/McGraw-Hill, (1991).

[9] "The C programming language". Brian W. Kernighan, Dennis M. Ritchie. Prentice Hall PTR, (2006).

[10] "Lenguaje C". Francisco Javier Moldes Teo. Anaya Multimedia, (2006).

[11] "Técnicas avanzadas de diseño de software: Orientación a objetos, UML, patrones de diseño y Java". José F. Vélez Serrano, Ángel Sánchez Calle, Alfredo Casado Bernárdez, Santiago Doblas Álvarez. Universidad Rey Juan Carlos, formato digital, (2009).

[12] "Diseño orientado a objetos con UML". Raúl Alarcón. Grupo EIDOS, (2000).

[13] "LAM/MPI Parallel Computing": http://www.lam-mpi.org/

[14] "MPICH: High-performance and Widely Portable MPI": http://www.mcs.anl.gov/research/projects/mpich2/

[15] "OPEN MPI: Open Source High Performance Computing": http://www.open-mpi.org/

[16] "Pov-Ray – The Persistence of Vision Ray-tracer": http://www.povray.org/