Universidad Rey Juan Carlos
I. T. Informática de Sistemas

Algoritmo de equilibrio de carga en un cluster heterogéneo

DIEGO CHAPARRO GONZÁLEZ
TUTOR: JOSE LUIS BOSQUE ORERO

INDICE:

INTRODUCCIÓN:

1.1.- Sistemas distribuidos:

Los sistemas distribuidos cada vez tienen mayor relevancia y comienzan a sustituir a los potentes mainframes. La aparición de estos sistemas se debe a dos avances tecnológicos. Uno de ellos es el desarrollo de los microprocesadores, que cada vez son más potentes y que hacen que ya no se cumpla una ley que hasta ahora si se cumplía con los mainframes:

"la potencia de CPU es proporcional al cuadrado de su precio"

Y el otro avance es el desarrollo de las redes de área local de alta velocidad, que proporcionan un medio de comunicación entre docenas o cientos de ordenadores con una velocidad cada vez mayor.

Estos sistemas distribuidos se forman mediante una red (usualmente una LAN) y un conjunto de CPU's(nodos). Éste tipo de sistemas presentan muchas ventajas con respecto a los mainframes:

- **ECONÓMICA**: Mejor relación precio/rendimiento. Ya que a un precio inferior al de un mainframe, se consigue un rendimiento muy alto.

- **VELOCIDAD**: Pueden alcanzar un mayor nivel de cómputo que los mainframes.
- **ESCALABILIDAD**: Son fácilmente escalables. Simplemente hace falta añadir una nueva CPU.
- **ESTABILIDAD**: Mayor estabilidad. Ya que aunque falle una de las CPU's, el sistema sigue funcionando.

Para conseguir que los sistemas distribuidos alcancen un mayor nivel de rendimiento hay que tener en cuenta también los problemas que pueden presentar:

- **SOFWARE**: Todavía no existe software suficientemente desarrollado para este tipo de sistemas.
- **REDES DE INTERCONEXIÓN**: Las redes pueden llegar a saturarse y causar otro tipo de problemas, que no ocurren en mainframes.

1.2.- Distribución de carga:

Uno de los principales aspectos para obtener un buen rendimiento de estos sistemas es la distribución de carga.

Es un componente muy importante, ya que de él depende el buen uso que se hace de la capacidad global de rendimiento del sistema (**eficiencia**). El rendimiento global depende en gran medida del algoritmo elegido para la distribución de carga.

La distribución de carga es el método para que los sistemas distribuidos obtengan el mayor grado de eficiencia posible. La distribución de carga consiste en el reparto de la carga entre los nodos del sistema para que la eficiencia sea mayor. La carga se distribuye y se traslada de los nodos más cargados a los nodos menos cargados del sistema. Con esto se consigue una reducción del tiempo de ejecución de

las tareas en el sistema distribuido, y se consigue aproximar el tiempo de finalización de la ejecución de las tareas de cada uno de los nodos, es decir, se consigue que los nodos terminen de ejecutar *"al mismo tiempo"*, y que no haya grandes diferencias entre el término de uno de ellos y él de los demás. Con todo esto se consigue que no haya nodos sobrecargados, y otros libres de trabajo, porque estos nodos sobrecargados trasladarían parte de su trabajo a esos otros nodos más libres de trabajo.

Los algoritmos de distribución de carga se basan en unos componentes, que marcan las diferencias entre unos algoritmos y otros:

- **Política de información**: La cuál decide qué tipo de información debe recoger de los nodos, de qué nodos debe recoger ésta información, y cuándo debe recogerla.
- **Política de transferencia**: Decide si cada uno de los nodos es apto o no para llevar a cabo una transferencia, tanto de emisor como de receptor.
- **Política de selección**: Decide cuál es la carga que se va a transferir en una transferencia.
- **Política de localización**: Localiza el nodo adecuado para realizar una determinada transferencia.

De éste aspecto depende en gran medida el rendimiento global, ya que dependiendo de la elección del algoritmo de distribución, se puede conseguir que todos los nodos del sistema estén trabajando en todo momento para minimizar el tiempo de ejecución de los procesos, o una mala elección del algoritmo de distribución puede hacer que tan solo unos pocos nodos estén sobrecargados mientras que otros no tengan nada de trabajo.

Esa es la función principal que debe llevar a cabo el algoritmo de distribución de carga, hacer que todos los nodos estén trabajando mientras que haya

algún proceso pendiente, y sea posible ejecutar dicho proceso en varios nodos consiguiendo una disminución en el tiempo de proceso. Hay algunos casos en los que el algoritmo de distribución no debe repartir carga entre los demás nodos ya que el tiempo de ejecución de dicho proceso puede ser tan pequeño, que si se repartiera entre los nodos se perdería demasiado tiempo en la comunicación y el resultado sería un tiempo de ejecución mayor.

El algoritmo de distribución debe tener en cuenta todos estos detalles. Pero además existen varios políticas que hay que definir para llegar a elegir un algoritmo de distribución eficiente:

- **Centralizado vs Distribuido:**

  Las políticas centralizadas son aquellas en las que la información se concentra en una única ubicación física, que toma todas las decisiones de planificación. Esta solución presenta problemas de cuellos de botella, y tiene un límite en su grado de escalabilidad.

  Por otro lado las políticas distribuidas son aquellas en las que la información está repartida entre los distintos nodos, y las decisiones son tomadas entre todos. Los problemas son que la información puede no ser coherente, se replica la información y necesita más comunicaciones.

- **Estáticas vs Dinámicas:**

  Las políticas estáticas toman decisiones de forma determinista o probabilística sin tener en cuenta el estado actual del sistema. Esta solución puede ser efectiva cuando la carga se puede caracterizar suficientemente bien antes de la ejecución, pero falla al ajustar las fluctuaciones del sistema.

  Las políticas dinámicas utilizan información sobre el estado del sistema para tomar decisiones, por lo que potencialmente mejoran

a las políticas estáticas mejorando la calidad de las decisiones. Incurren en mayor sobrecarga al tener que recoger información de estado en tiempo real.

Todos estos conceptos y otros más son los que habrá que tener en cuenta para llegar a diseñar un algoritmo de distribución de carga, para un sistema distribuido, que consiga obtener el mayor rendimiento global posible de dicho sistema.

1.3.- Objetivos:

Los objetivos fundamentales para la realización del presente proyecto son los siguientes:

1. Estudiar los sistemas distribuidos, y las aplicaciones paralelas desarrolladas sobre ellos.
2. Realizar un estudio de una librería que permite el paso de mensajes para realizar aplicaciones paralelas (MPI).
3. Estudiar diferentes algoritmos de distribución de carga.
4. Diseñar un algoritmo de distribución de carga **dinámico** y **distribuido**.
5. Estudio de una aplicación CBIR (Content-Based Image Retrieval).
6. Implementación de un algoritmo de distribución de carga dinámica y distribuido sobre una aplicación CBIR.
7. Obtener unos resultados, para su posterior comparación con otros algoritmos de distribución de carga.

ESTADO DEL ARTE:

2.1.- Introducción:

En la actualidad existen varios algoritmos utilizados para el equilibrio de carga en sistemas distribuidos. El esquema general de los conceptos que distinguen unos algoritmos de otros es el siguiente:

- **Dinámico** o **estático**.
- **Centralizado** o **distribuido**.
- **Política de Información**:
  - Bajo demanda.
  - Actualización periódica.
  - Recogida por cambios en el estado del sistema.
- **Política de transferencia**:
  - Basada en umbral o en transferencia relativa.
  - Periódica o disparada por eventos.
  - Iniciada por el emisor o por el receptor.
- **Política de transferencia**:
- **Política de localización**.

Éstos algoritmos son los siguientes:

2.2.- Algoritmos Centralizados:

En [1] se propone un algoritmo denominado CENTRAL. En éste algoritmo un nodo es emisor cuando su carga excede un cierto umbral T. Sólo considera seleccionables aquellos procesos con tiempo de ejecución mayor que otro umbral. Si no se cumplen estas condiciones de trabajo se ejecuta localmente (obliga a conocer el tiempo de ejecución con anterioridad). Periódicamente todos los nodos envían su información de carga a un nodo central que la recoge y toma las decisiones de distribución. Cuando el nodo central recibe una solicitud de transferencia selecciona el nodo con menor longitud de cola y le informa al solicitante, quien se encarga de la transferencia. Concluyen que para un número de nodos limitado y medio de comunicación eficiente esta solución es sencilla y eficiente.

En [2] se propone una estrategia similar pero el número de nodos que envían mensajes de actualización se restringe, enviando un valor de corte de carga.

Muchos autores afirman que los algoritmos centralizados no son escalables, porque el nodo central es un cuello de botella potencial. La capacidad funcional de cualquier servidor centralizado es limitada y no puede crecer de forma ilimitada[4].

2.3.- Algoritmos de clustering distribuidos:

En [3] se propone un algoritmo parametrizado no intrusivo para sistemas distribuidos. La sobrecarga que se induce en cada recurso es menor que un límite prefijado por el dueño del recurso. Los nodos se agrupan en clusters, y a cada uno de estos clusters se le asigna un gestor para mantener el estado del cluster. El tamaño de un cluster es un parámetro determinado con respecto a la escalabilidad y a las medidas de intrusividad del sistema. El conjunto de gestores de los clusters puede cambiar en el tiempo para manejar los límites de la intrusividad. Los nodos envían actualizaciones y solicitudes de tareas al gestor de su cluster. Cuando un gestor recibe una solicitud que no puede satisfacerse dentro del cluster pregunta a los otros

gestores. Mantener el estado global coherente minimizaría estas preguntas pero introduce problemas de escalabilidad. Cada gestor mantiene una lista de clusters disponibles para garantizar que se cumplen los límites de intrusividad. Cuando un gestor detecta que está en peligro de violar las restricciones de intrusividad se borra de la lista e informa a los demás.

2.4.- Autómatas de Aprendizaje automático:

El aprendizaje del autómata se basa en una política probabilística. Se asigna una probabilidad a cada una de las posibles acciones (vector de probabilidad). Las acciones son cosas como: “manda tarea a al nodo J”. Inicialmente estas probabilidades son iguales, porque a priori no se conoce la optimalidad de cada acción. Después de ejecutar una acción se recibe una respuesta del destino indicando si la acción es buena o mala, y en función de ella se modifica el vector de probabilidad. El aprendizaje del comportamiento del autómata es el siguiente:

- Si la respuesta es favorable para una determinada acción, se incrementa la probabilidad de ésta.
- Si la respuesta no es favorable, se decrementa la probabilidad de esta acción.

Las principales ventajas son las siguientes[8]:

- El vector de probabilidad modela de forma sencilla la historia de las respuestas del entorno.
- El esquema de aprendizaje puede refinarse para mejorar el rendimiento y evitar inestabilidad.
- No se tienen que realizar costosos cálculos para tomar una decisión.

2.5.- Política de localización aleatoria:

Una posible política de localización aleatoria es la siguiente. [7] Un nodo se considera emisor si la longitud de su cola de CPU excede un umbral. El nodo destino se selecciona aleatoriamente entre todos. No hay intercambio de información de estado, por lo que es un algoritmo no cooperativo. Una cuestión importante es cómo trata el nodo destino la nueva tarea. Si se considera como una tarea ordinaria, puede reenviarla en caso de estar sobrecargado. Esto se llama “aleatorio incontrolado” y es inestable si el coste de la transferencia es mayor que cero. No importa la carga media, hay una probabilidad positiva de que todos los nodos transfieran a la vez y ninguno procese las tareas. La inestabilidad se puede evitar restringiendo el número de veces que una tarea puede ser transferida, mediante un límite de transferencia. El rendimiento de esta política es muy alto. Esta política es muy escalable, ya que no recoge ninguna información.

En [5, 6] se presenta una solución similar en la cual los nodos destino se seleccionan aleatoriamente de un subdominio del sistema.

2.6.- Algoritmo de sondeo:

Política iniciada por demanda; un nodo dispuesto a participar en una transferencia sondea a los demás para encontrar un socio adecuado. Las pruebas se pueden hacer en serie o en paralelo. Un nodo puede seleccionarse para el sondeo aleatoriamente, a partir de información recogida en pruebas anteriores, o como vecino más próximo. [7] propone un algoritmo que selecciona aleatoriamente y se prueba si la transferencia dejaría al receptor por encima de un umbral. Si esto ocurre se busca otro nodo; si no se transfiere la tarea. Este proceso se repite hasta encontrar un nodo adecuado o hasta superar un límite de pruebas. El objetivo de esta política es evitar transferencias inútiles.

Esta política funciona mejor que la aleatoria, lo que sugiere que la sobrecarga de recoger alguna información de estado es superada por los beneficios. En [7] se prueba en paralelo un conjunto de nodos seleccionados aleatoriamente y se selecciona el mejor. El rendimiento de esta política no es mucho mejor que la anterior. Otras soluciones incluyen mantener alguna información de estado de los nodos, obtenida en pruebas previas o de diseminación con poca frecuencia.

2.7.- Algoritmos basados en ofertas:

Utilizan un protocolo de contrato con tres fases. En la primera un nodo dispuesto a transferir parte de su carga difunde una solicitud de ofertas. Recibe las ofertas de los nodos dispuestos a aceptar carga. Evalúa las ofertas y envía la carga al nodo que envió la mejor oferta. En [6] hacen varias rondas de mensajes antes de aceptar definitivamente la tarea. Esto se hace porque un nodo descargado puede contestar a varias solicitudes, ser seleccionado por varios como receptor y sobrecargarse. Es posible que un nodo considere las ofertas que ha hecho antes cuando evalúa la contestación a otra solicitud. Puede mantenerse información de estado de los nodos, basada en ofertas anteriores, que se usa para seleccionar el subconjunto de nodos al que se le envía la solicitud de oferta. En [5] se propone un algoritmo de ofertas adaptativo: envía la solicitud sólo a aquellos nodos con distancia menor que d. La distancia d se incrementa si no se reciben suficientes ofertas o se decrementa si se reciben demasiadas, en un intervalo de tiempo que también depende de d. Estos algoritmos también se utilizan en sistemas de tiempo real .

2.8.- Algoritmo de reclutamiento:

[9] Para reducir los mensajes del algoritmo anterior se propone el siguiente algoritmo. Definen tres posibles estados para un nodo: ligero (L), que puede aceptar

tareas; pesado (P) puede transferir tareas y normal (N) no necesita esfuerzos de distribución. Cada nodo tiene una tabla de carga con la información de todos los procesadores candidatos. Debido a retrasos en la comunicación las tablas pueden contener diferente información del mismo modo. El conjunto de candidatos es independiente de la implementación.

Cuando un nodo es ligero envía un mensaje de solicitud de reclutamiento a todos los nodos pesados y estos contestan con un mensaje de edad del recluta. Si lleva poco tiempo pesado responde con un 0. La edad puede definirse como máximo número de procesos en espera, número de procesos, etc. Después de recibir todas las respuestas o pasado un intervalo de tiempo, el nodo origen calcula un draft-estandar basado en las respuestas recibidas. Si este nodo todavía tiene un proceso cuya edad es menor que la calculada, lo transfiere, sino se envía un mensaje “demasiado tarde”. Cada proceso se espera que migre una sola vez. Si los nodos cambian de estado frecuentemente entre H y N y entre N y L se generan muchos mensajes. Hay técnicas para reducir estos mensajes enviando sólo los que son interesantes. Este algoritmo alivia muchos problemas de los algoritmos basados en ofertas [5].

2.9.- Algoritmo de Compartición de Carga Flexible:

FLS divide el sistema en dominios. Un nodo solo intercambia información y carga con los de su dominio. Un nodo determina que otros pertenecen a su dominio en función de su estado: sobrecargado (O), infrautilizado (U) o media carga (M). La membresía de los dominios es simétrica: si A está en el dominio de B, B está en el de A. Cuando un nodo cambia de estado informa los demás del dominio y se borra. La elección de los miembros se basa en mensajes con información de estado y también se hace periódicamente. El tamaño del dominio está limitado. Debido a retardos de comunicación, la información de estado puede estar obsoleta. Los miembros del dominio son sólo una pista y se necesita un protocolo de solicitud respuesta que permita rechazar nuevas tareas. Si un nodo descargado rechaza una tarea se borra del dominio de origen y la tarea se ejecuta localmente.

2.10.- Algoritmo Adaptativo Iniciado Simétricamente:

Los nodos se clasifican como emisores, receptores y OK, en función de información obtenida de pruebas previas. Cada nodo mantiene tres listas con el estado de los otros nodos: lista de receptores, lista de emisores y lista de OK. Inicialmente cada nodo asume que todos los demás son receptores. Todos pueden iniciar transferencias como emisor o receptor. El componente emisor de la política de localización se dispara cuando la carga del nodo excede un umbral y probará el primero de la lista de receptores. Este pondrá al origen el primero en su lista de emisores, y contestará con un mensaje indicando su estado actual. Al recibir este mensaje el emisor transferirá una tarea si la respuesta indica que puede aceptarla. Si no lo borrará de la lista de receptores y lo pondrá el primero de la lista adecuada. Las pruebas acaban cuando se encuentra un receptor adecuado, el número de intentos llega a un límite o la lista de receptores está vacía. El componente iniciado por el receptor selecciona nodos para probar primero de la lista de emisores del primero al último, usando la información más actualizada primero. Si esta lista se vacía buscará en la lista de OK y en la de receptores, pero desde la cola a la cabeza usando la información más desactualizada primero. El resto del algoritmo es similar al caso anterior.

2.11.- Difusión Natural:

Solo utiliza información local y consigue un equilibrio global. Los procesadores sobrecargados distribuyen parte de su carga a sus vecinos inmediatos. El intercambio de mensajes se hace según un grafo no dirigido G= (V, E), V nodos. La parte de carga que se desplaza del nodo X al Y se describe mediante una doble matriz de difusión estocástica P^G, con P(X, Y) >0 si A_G (X, Y) >0, donde A_G denota

la matriz de adyacencia de G. Para un P_G adecuado converge a una distribución uniforme.

La difusión podría ser lenta dependiendo de P_G y de la conectividad de G. Otra solución es la difusión parcial en la cual el grafo de equilibrio de carga se divide en partes que pueden intersecarse. La carga se equilibra haciendo un paso de equilibrio en cada división. Si hay muchas particiones la tasa de convergencia es superior a la difusión natural.

SOLUCIÓN PROPUESTA:

3.1 Introducción:

Se ha desarrollado un algoritmo de distribución de carga dinámico y distribuido. Éstas son dos características muy importantes para estos algoritmos ya que las implementaciones de este tipo de algoritmos no están muy desarrolladas. Otros algoritmos que son centralizados y/o estáticos tienen más implementaciones que el desarrollado en este proyecto, ya que su complejidad es menor.

La característica de ser **distribuido** le proporciona unas ventajas que no se presentan en los algoritmos centralizados. Permite una gran escalabilidad, es decir, un crecimiento del número de nodos del sistema, sin que el sistema de distribución reduzca su eficiencia. Además elimina los cuellos de botella que se producen en los sistemas centralizados, por tener todos los datos en un solo nodo y porque todos los demás nodos deben acceder a éste para el desarrollo de la aplicación. Además de esto, los algoritmos distribuidos son más tolerantes a fallos, ya que no existe ningún nodo que sea crítico para el buen funcionamiento de la aplicación.

La otra característica es que es **dinámico**, lo cual también nos presenta varias ventajas adicionales sobre los algoritmos estáticos. El hecho de ser dinámico permite al algoritmo adaptarse a la situación del sistema. El sistema se amolda a la carga existente en cada momento, y permite realizar la distribución cuando cambia el

estado del sistema y no solamente realizar una distribución inicial, como hacen los algoritmos estáticos.

Este algoritmo pretende dar solución a uno de los problemas que se pueden plantear en las aplicaciones distribuidas sobre un sistema distribuido. Si la ejecución de dichas aplicaciones no cuenta con un sistema de distribución de carga, el tiempo de respuesta puede ser muy variable dependiendo del estado de cada uno de los nodos del sistema.

Cuando el sistema está sobrecargado y cada uno de sus nodos también no se puede utilizar ninguna solución para mejorar notablemente el tiempo de ejecución de las aplicaciones distribuidas, ya que todos los nodos estarían sobrecargados y no habría manera de utilizar tiempo de CPU adicional de ninguno de los nodos.

Pero cuando el sistema no está sobrecargado, y alguno de los nodos si lo está puede presentar un gran problema a una aplicación distribuida que utilice ese nodo para ejecutar, y no posea ningún algoritmo de distribución de carga. El tiempo de respuesta de dicha aplicación sería como mínimo el tiempo de respuesta del nodo que más tardase en ejecutar (es decir, el más sobrecargado).

La solución a este problema consiste en utilizar un **algoritmo de distribución de carga**. Este algoritmo se debe encargar de repartir la carga en el sistema distribuido para que al ejecutar las aplicaciones distribuidas no haya nodos sobrecargados y otros poco cargados, sino que se reparta la carga para que el tiempo de respuesta de todos los nodos sea lo más similar posible.

El algoritmo desarrollado pretende dar solución a este problema. Este algoritmo se encarga de realizar la distribución de carga de una aplicación de tipo CBIR (Content Based Information Retrieval System). Este tipo de aplicaciones consta de una gran cantidad de datos de naturaleza multimedia, en nuestro caso

imágenes, y la finalidad principal de dichas aplicaciones es el almacenamiento y gestión eficiente de dichos datos. En nuestro caso la aplicación consiste en buscar el número n de imágenes más parecidas a una imagen que se presenta como entrada de la aplicación, y la búsqueda se realiza sobre una base de datos de 30 millones de imágenes.

Lógicamente, debido al gran coste de procesamiento de dicha aplicación, ésta es distribuida. Y el objetivo fundamental del algoritmo desarrollado en el presente proyecto es la distribución dinámica y distribuida de la carga presentada por esta aplicación.

3.2 Principales características:

El sistema distribuido usado es un cluster de 26 PCs. 25 de ellos actuarán como slaves de la aplicación CBIR(Capítulo 4.1), y otro PC servirá como master de dicha aplicación.

A continuación se describen brevemente las características principales del algoritmo desarrollado, que más adelante serán explicados con mayor detalle:

- **Dinámico**: el algoritmo parte de un reparto uniforme de la carga entre todos los nodos. Y cuando un nodo acaba el trabajo con la carga que tiene obtiene más carga de otro de los nodos (el más sobrecargado).
- **Distribuido**: no hay ningún proceso central que se encargue de realizar la distribución en todos los nodos. Sino que en cada nodo hay un proceso que se encarga de esta labor.
- Política de Información: utiliza una política **periódica** para definir el estado del sistema.

- Política de transferencia: en este caso es **iniciada por el receptor** y está basada en una transferencia **relativa**, y en **umbrales**.
- Política de selección: se calcula de forma **relativa el estado** de ambos nodos.
- Política de localización: se basa en la **información** que tiene cada nodo del **estado del sistema**.

3.3.- Conceptos Básicos:

Existen una serie de conceptos básicos que se han utilizado para el desarrollo del algoritmo, y que se usarán para describir detalladamente el mismo en el apartado siguiente:

3.3.1.- Carga de trabajo:

La carga de trabajo será el nº de imágenes que tiene que comparar y cada imagen es una unidad de carga (Capítulo 4.1).

3.3.2.- Firma:

Corresponde a una forma de representar una imagen. Se halla a partir de la imagen aplicándole una serie de transformadas. En el presente documento se ha utilizado indistintamente la palabra imagen y firma.

3.3.3.- Velocidad inicial:

Se ha calculado la velocidad inicial de cada nodo, ejecutando el proceso solver en vacío, es decir sin implementación paralela y sin carga en la máquina, de forma local en cada uno de ellos, y midiendo sus tiempos de ejecución. Es decir, es la capacidad de proceso de un nodo en vacío, sin ninguna carga adicional.

3.3.4.- Carga local:

La carga local indica el estado de cada nodo. Se calcula a partir del número de procesos en ejecución que hay en cada máquina.

3.3.5.- Velocidad relativa:

Se calcula a partir de la velocidad inicial, y de la carga local de un nodo. La velocidad relativa es igual a la división entre la velocidad inicial y la carga local del mismo.

3.3.6.- Tabla de estados:

Cada nodo tiene una tabla en la que almacena diversos datos sobre los demás nodos, así como el tiempo de ejecución que lleva con una determinada cantidad de carga, la carga de trabajo que tiene ese nodo, la carga local del nodo, la velocidad inicial, ...

3.3.7.- Umbral:

Es el valor mínimo de carga de trabajo que se puede transferir de un nodo a otro.

3.4.- Descripción del algoritmo:

El algoritmo desarrollado es un algoritmo completamente distribuido, dinámico e iniciado por el receptor. A continuación se detallan cada una de las políticas utilizadas en el algoritmo.

3.4.1.- Política de Información:

Se utiliza una política de **actualización periódica**. Este tipo de política puede presentar algún tipo de problemas debido al continuo paso de mensajes entre los nodos, pero también tiene sus ventajas. Y por estas ventajas es por lo que se ha elegido este tipo de política. La principal ventaja es que ofrece un **tiempo de respuesta muy bueno**.

Cuando se realiza una petición de carga para saber cuál es el nodo adecuado para realizar la transferencia, la respuesta es inmediata ya que en todo momento los datos que tienen todos los nodos sobre la información del estado de los demás están actualizados.

Para que todos los nodos tengan información sobre el estado de los demás hay que buscar una serie de índices que nos den una estimación de cuál es dicho estado. En el presente algoritmo se ha decidido utilizar como índice el número de **procesos** que hay **en ejecución**. Éste dato es el que se intercambian periódicamente todos los nodos para poder tener una información de estado actualizada de todos los demás. Este proceso de información se realiza de forma periódica, y este periodo de tiempo se ha calculado de forma experimental, se han hecho pruebas y se ha comprobado si este proceso consumía muchos recursos, y se ha llegado a la conclusión de que el periodo adecuado para realizar esta labor de información es de 10 segundos.

Aparte de este dato, cada nodo posee otros de los demás nodos:

Uno de ellos es la velocidad inicial (Capítulo 3.3.3) que nos servirá para ir calculando otros datos de interés, y para saber el estado de los demás nodos. Otro es la cantidad de carga de trabajo total que tienen los otros nodos. Esta información se obtiene al comienzo de la ejecución de la aplicación y cada vez que cambia este dato

en alguno de los nodos. Y otra información, además muy importante, que cada nodo tiene sobre los demás es el porcentaje de trabajo realizado, es decir, el número de unidades de carga local que ha ejecutado el solver (Capítulo 4.1). La forma de obtener este dato se realiza usando la información que se ha mencionado anteriormente, es decir, la carga local, la velocidad inicial y la carga de trabajo total. Además de ésta, se posee otra información que también se almacenan como el tiempo de última actualización y el tiempo total de ejecución con esa carga de trabajo.

Las actualizaciones del **número de tareas** se realizan de la siguiente manera:

$\mathbf{N_t}$ = número de tareas

T = Tiempo total de ejecución de ese nodo con esa carga

D = Diferencia de tiempo entre la última actualización y el tiempo actual.

$$\mathbf{N_t = ((N_t * T) + (N_t * D)) / (T + D)}$$

Lo que se hace es ir hallando la media, para tener un dato que nos indique la situación del nodo durante todo el proceso.

Para actualizar el **porcentaje** de carga de trabajo realizado en cada nodo se realiza como sigue:

V = Velocidad inicial del nodo

C = Carga total del nodo, en número de unidades de carga

P = Porcentaje realizado por el nodo

$$P = (((V * T) / C) * 100) / N_t$$

Lo que se hace es calcular el porcentaje realizado a partir de la velocidad y del tiempo que ha transcurrido, y luego dividirlo por el número de tareas, para que se corresponda con lo realizado dependiendo de la carga local.

3.4.2.- Política de Transferencia:

La decisión de si un nodo es adecuado para participar en una transferencia de carga se toma a partir de unos parámetros, que son:

- **Carga local de los nodos**. A partir de esta carga se calcula si es adecuado realizar una transferencia o no lo es. Se calculan los estados de ambos, receptor y emisor, y se comprueba si es adecuado.
- Está basada en **umbrales**. No basta con que la carga local de un nodo permita la transferencia de carga, sino que también es necesario que la cantidad de carga a transferir sea superior a un determinado umbral, definido a priori a partir de resultados experimentales. Si la carga a transferir no es suficiente, entonces el nodo no es apto para realizar la transferencia.

En todos los casos la transferencia siempre es **iniciada por el receptor**. Cuando ha acabado su trabajo, pide más carga a otro nodo.

3.4.3.- Política de selección:

Para calcular la cantidad de carga a transferir también influyen diversos parámetros. El primero de ellos es la carga local, que junto con la velocidad inicial (Capítulo 4.1), nos servirá para hallar la **velocidad relativa** de cada nodo:

Velocidad relativa = velocidad inicial / carga local

Con esto obtenemos las velocidades relativas en cada uno de los nodos.

También necesitamos la cantidad de carga de trabajo que le queda pendiente al proceso emisor. Para poder hallar a partir de ésta el porcentaje de carga que debería ser transferida al receptor. Y con estos datos ya podemos hallar la cantidad de carga que se debería transferir.

Primero debemos **hallar el porcentaje de carga que debería ser transferido al receptor**, dependiendo, claro está, de las velocidades relativas de ambos nodos:

P = Porcentaje de carga que se debería transferir.

V_r = Velocidad relativa del receptor.

V_e = Velocidad relativa del emisor.

$$\mathbf{P = (V_r / (V_r + V_e))}$$

Y una vez que tenemos el porcentaje, para hallar la **cantidad de carga de trabajo** solo nos queda multiplicar dicho porcentaje por la cantidad de carga pendiente en el nodo emisor:

C_p = Cantidad de carga pendiente en el emisor.

C = Carga que se debería transferir.

$$C = P * C_p$$

Esta carga hallada sería la carga ideal que se debería transferir para que tanto el proceso emisor como el receptor terminaran de procesar su carga al mismo tiempo. Pero hay otros factores en la aplicación que se ha paralelizado (solver, ver Capítulo 4.1), que hay que tener en cuenta para que esto sea efectivo.

Al terminar de ejecutar el solver, realiza una ordenación de los resultados y el tiempo que tarda también hay que tenerlo en cuenta para calcular la cantidad de carga. Por eso al cálculo anterior se le realizan algunas modificaciones para adaptar esta cantidad de carga a la situación real de la aplicación.

Hay que calcular el tiempo que tardaría el emisor en realizar la ordenación de resultados dependiendo de la cantidad de carga que le quedaría al realizar la transferencia, y el tiempo que tardaría el receptor en ordenar la carga que le sería transferida.

Se parte de un cálculo experimental del tiempo de ordenación, y se ha llegado a la conclusión que en 1 segundo se ordenan 10.000 unidades de carga. A partir de esto obtenemos los **tiempos de ordenación de ambos nodos**:

T_e = Tiempo que tardaría el emisor en ordenar la carga resultante.

C_t = Carga de trabajo total que tiene el emisor.

T_r = Tiempo que tardaría el receptor en ordenar la carga transferida.

$$T_e = ((Ct - C) / 10.000)$$

$$T_r = (C) / 10.000$$

Y a partir de estos tiempos y de la velocidad relativa se calcula la **cantidad de carga que realizaría el emisor y el receptor en esos tiempos** y se modifica la cantidad de carga calculada anteriormente.

C_{et} = Carga que realizaría el emisor en T_e

C_{rt} = Carga que realizaría el receptor en T_r

$$C_{et} = T_e * V_e$$

$$C_{rt} = T_r * Vr$$

Y finalmente a la **carga** calculada anteriormente, se le suma C_{et} y se le resta C_{rt}.

$$C = C + C_{et} - C_{rt}$$

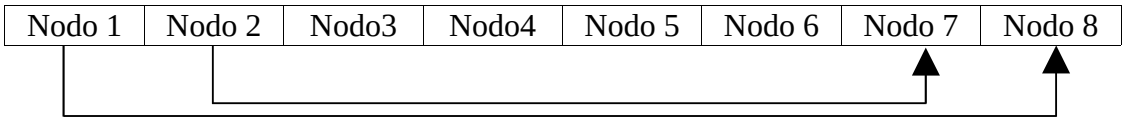
Con el cálculo de esta carga se pretende que tanto el receptor como el emisor terminen su ejecución al mismo tiempo.

3.4.4.- Política de localización:

Cuando un nodo busca un emisor para realizar una transferencia, utiliza los datos sobre el **estado de los demás nodos** que tiene almacenados, que ya se han comentado en los apartados anteriores.

La elección del nodo adecuado se realiza de la siguiente manera. El nodo emisor comprueba el estado de los demás nodos y verifica cual es la cantidad de carga de trabajo que le queda a cada nodo en unidades de carga. Con estos datos genera una **lista ordenada por cantidad de carga restante** de todos los nodos. Y una vez que ha creado esta lista la elección se realiza de la siguiente manera.

Comprueba cual es su posición, y elige **al simétrico en la lista**. Es decir, si por ejemplo el proceso 2 y el 1 buscan un nodo emisor:



El nodo 2 elegiría al nodo 7 y el nodo 1 elegiría al nodo 8. Esta solución para elegir al nodo emisor presenta bastantes ventajas. Al estar la lista ordenada por cantidad de carga de trabajo restante, se consigue que cuando un nodo termina de realizar su trabajo estará en un extremo de la lista, y elegirá como nodo emisor al nodo que esté en el otro extremo de la misma, que será uno de los que más carga de trabajo pendiente tenga por realizar. Con esto se consigue que la elección del nodo emisor sea muy coherente, ya que los nodos menos cargados le pedirán carga a los más cargados, mientras que los que estén en el medio, ni en un extremo ni en el otro realizarán su trabajo sin tener que recibir peticiones de carga.

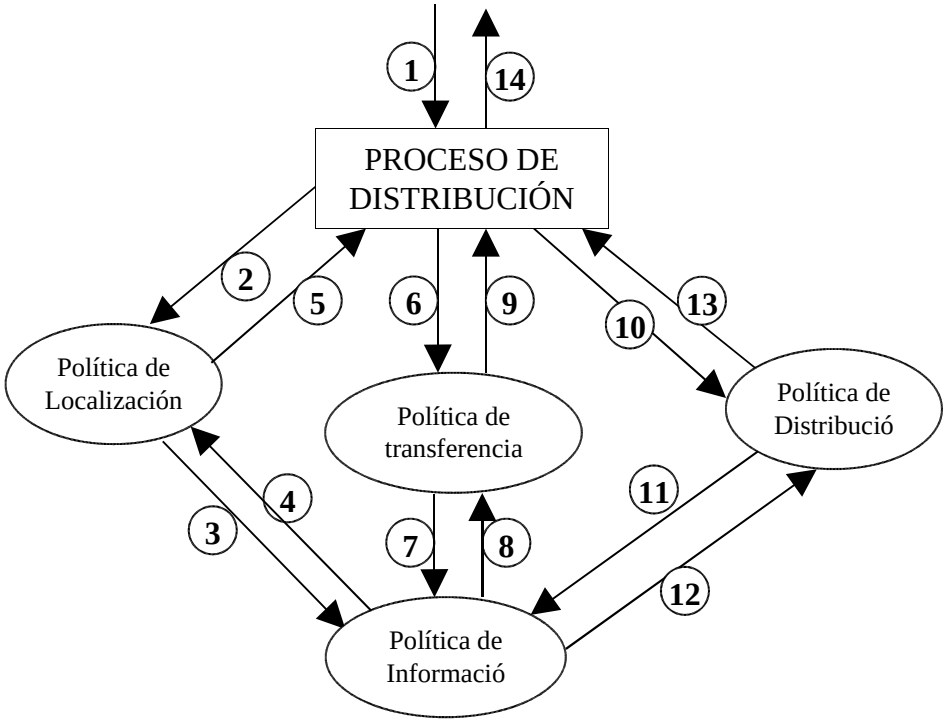
Además con esta solución se presenta otra gran ventaja. Si varios nodos están buscando un nodo emisor, en **ningún caso le realizarán peticiones al mismo nodo**, sino que elegirán los nodos con mayor carga de trabajo restante, pero no elegirán el mismo, y con esto se evita que un nodo pueda estar recibiendo mensajes continuamente mientras que otro no recibe ninguno. Se eliminan potenciales cuellos de botella.

Este proceso de elección no es costoso en tiempo, ya que los nodos tienen información actualizada del estado de todos los demás, y por ello solo deberán ordenar la lista y obtener el nodo correspondiente.

Si el nodo elegido para la transferencia no puede realizar la transferencia de carga, por ejemplo por que no pasa el umbral de carga necesario, se selecciona otro de la lista de los adyacentes a éste. Primero se eligen los que más carga restante tienen y después se avanza por la lista hacia los que menos carga restante tienen hasta que se encuentra un nodo adecuado, o hasta que se acabe la lista.

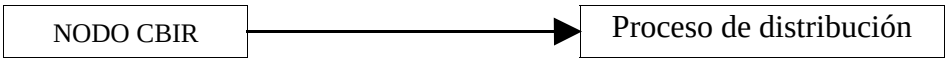
3.5.- Diagramas de flujo de datos:

A continuación se presenta un diagrama de flujo de datos que presenta a alto nivel, el funcionamiento del algoritmo. Se puede observar que funciones realiza el algoritmo de distribución y como utiliza cada una de las políticas anteriormente comentadas. Todo este proceso se lleva a cabo cada vez que debe entrar en acción el algoritmo de distribución, es decir, cuando uno de los nodos ha terminado de hacer su trabajo y pide carga adicional al proceso de distribución.

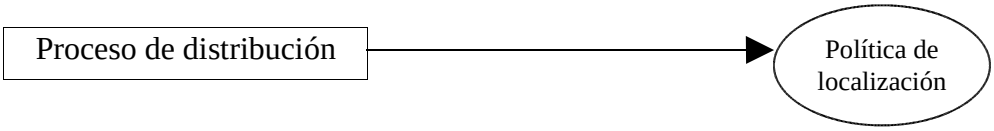


Los mensajes que se intercambian entre cada uno de los componentes son los siguientes:

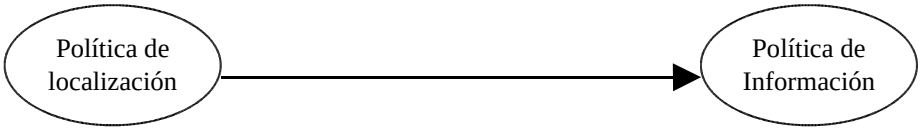
1. Un nodo CBIR, cuando ha terminado con su carga de trabajo, pide más carga al algoritmo de distribución.



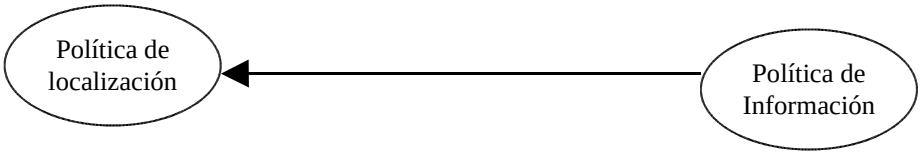
2. El algoritmo de distribución le consulta a la política de localización para saber a qué nodo hay que pedirle la carga para transferir.



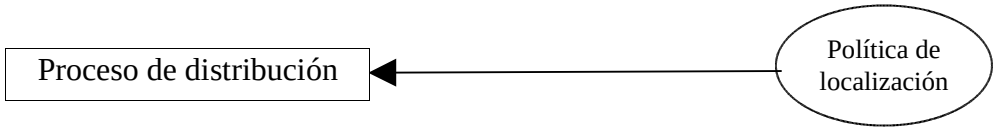
3. La política de localización consulta a la política de información para obtener el estado de los demás nodos.



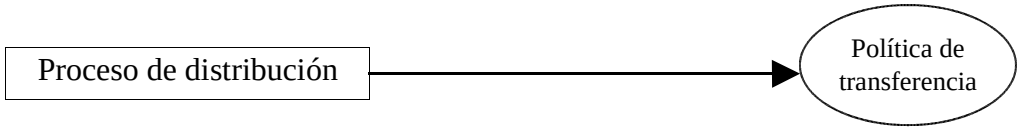
4. La política de información le indica a la política de localización el estado de los nodos.



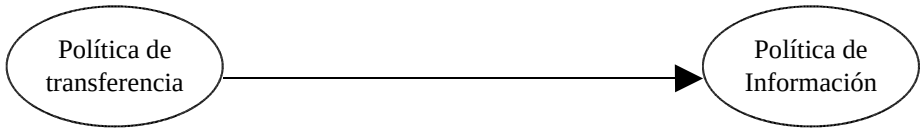
5. Una vez que tiene información sobre los demás nodos, la política de localización selecciona el nodo más adecuado, y se lo dice al proceso de distribución.



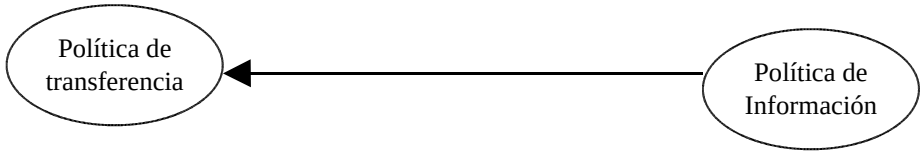
6. El proceso de distribución solicita a la política de transferencia que informe sobre si el nodo es adecuado para realizar la transferencia o no.



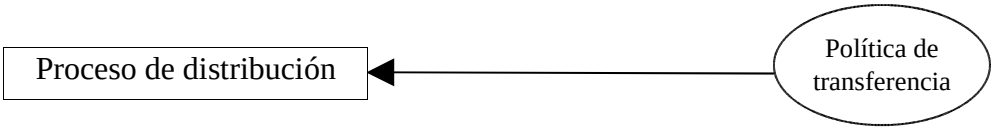
7. La política de transferencia solicita a la política de información la información de estado sobre los demás nodos.



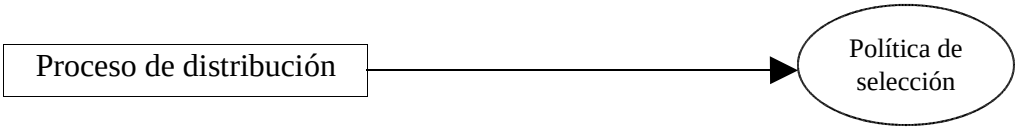
8. La política de información le devuelve a la política de transferencia de estado de los nodos.



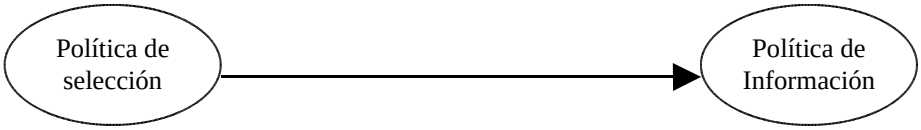
9. La política de transferencia verifica si el nodo es adecuado o no para la transferencia y le responde al proceso de distribución.



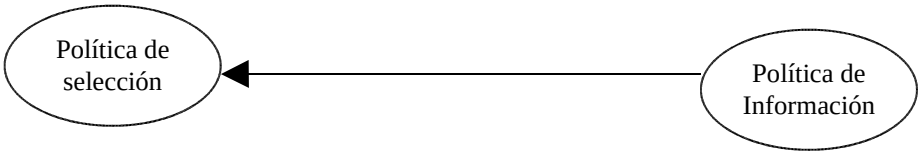
10. Si el resultado de la política de transferencia es afirmativo, entonces el proceso de distribución solicita que cantidad de carga debe ser transferida a la política de selección. Si el resultado es negativo, volvería a empezarse por el paso 2.



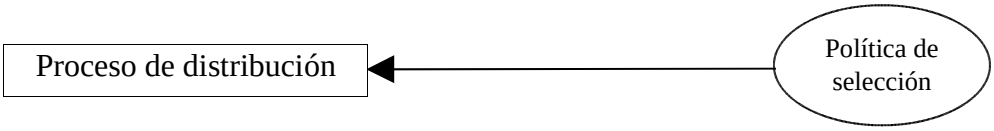
11. La política de selección solicita a la política de información el estado de los demás nodos.



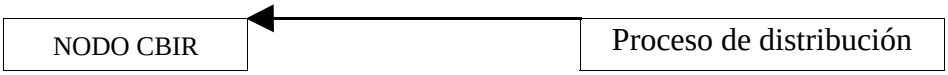
12. La política de información le responde a la política de selección y le informa del estado de los nodos..



13. La política de selección calcula la cantidad de carga que debe ser transferida y se lo indica al proceso de distribución.



14. El proceso de distribución obtiene la carga que debe ser transferida y se la entrega al nodo CBIR.



IMPLEMENTACIÓN:

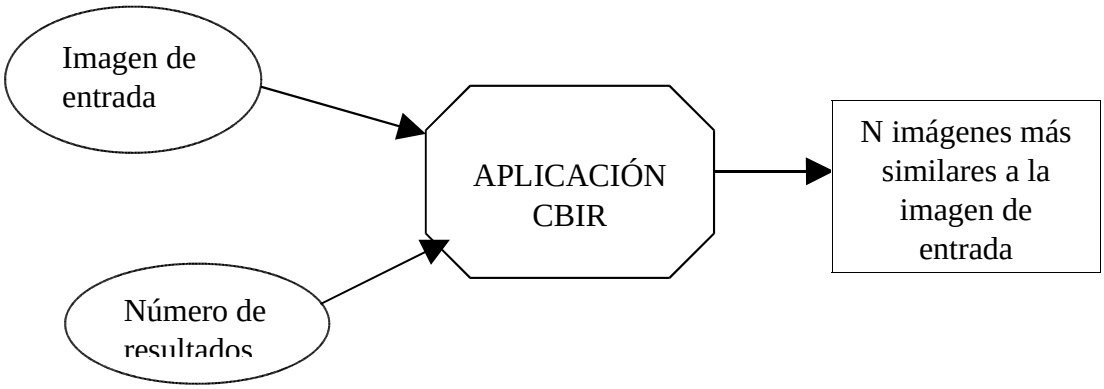
4.1.- Aplicación CBIR:

El algoritmo desarrollado como ya se ha comentado anteriormente realiza las labores de **distribución de carga sobre una aplicación distribuida**. Esta aplicación es una aplicación de tipo CBIR (Content Based Information Retrieval).

Este tipo de aplicaciones son sistemas de información, basados en una **gran cantidades de datos** de tipo multimedia, y cuya principal finalidad es el almacenamiento y la gestión eficiente de todos estos datos.

En nuestro caso la aplicación consiste en la gestión de una gran base de datos de imágenes, en la cual se debe buscar el número n de imágenes que más se asimilan a una imagen que se proporciona como entrada de la aplicación.

ENTRADAS: SALIDA:



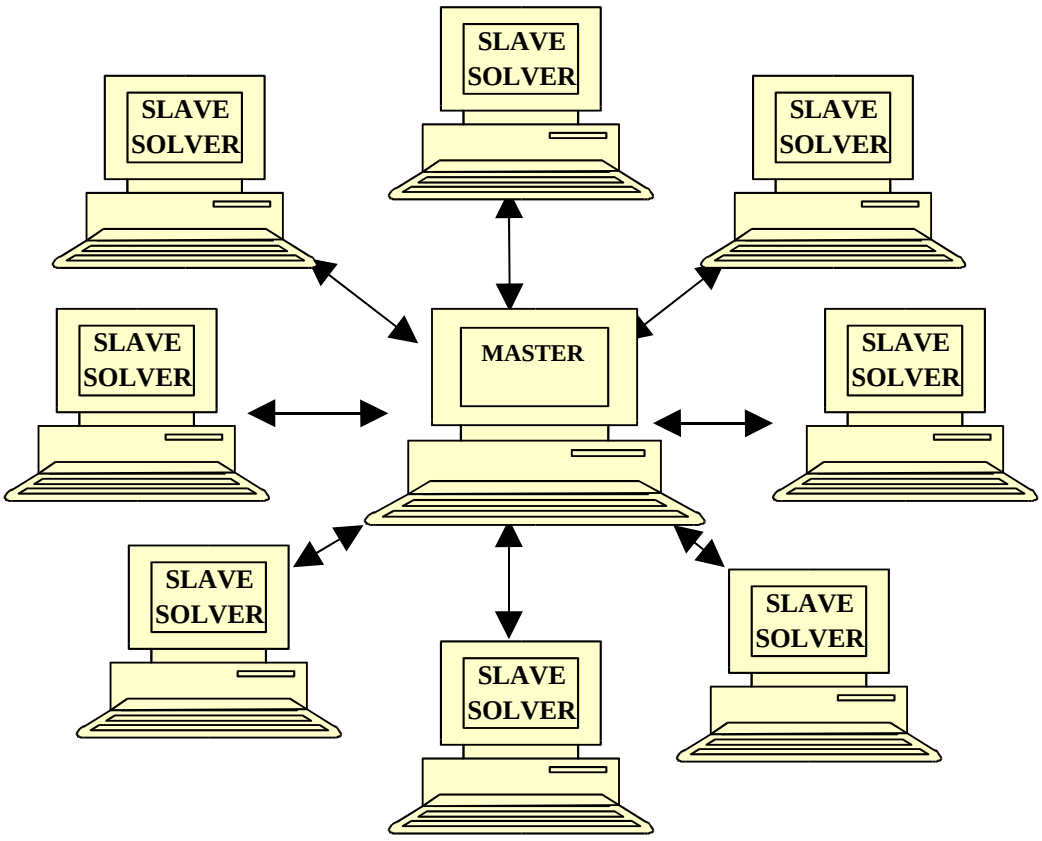
Esta aplicación requiere una gran cantidad de cómputo debido a la complejidad del algoritmo de comparación, y a la cantidad tan elevada de datos almacenados. En nuestro caso, la aplicación maneja una base de datos de 30 millones de imágenes.

Por todo esto, la aplicación debe tener una solución paralela para la obtención de buenos resultados. La estrategia de distribución de la aplicación está basada en un modelo del tipo granja (farm), en el que un proceso master distribuye los datos a procesar sobre un conjunto de procesos de tipo slave que se encargan de procesar los datos y devolver los resultados parciales al master cuando han terminado de procesarlos.

Cada uno de los procesos slave, se encarga de ejecutar un proceso denominado solver, que es en realidad el que realiza el procesamiento de las imágenes. El solver, después de procesar las imágenes ordena los resultados y obtiene las n imágenes más similares a la imagen de entrada, y el slave se las manda al master.

Esta distribución se realiza sobre **MPI** (Message Passing Interface), que es una librería que permite el paso de mensajes para la realización de aplicaciones paralelas.

El esquema de la distribución de la aplicación CBIR es la siguiente:



A partir de éste esquema se puede concluir que tiempo total de ejecución del sistema depende del tiempo de ejecución del nodo más lento, es decir el que más tarde en realizar su trabajo. El presente proyecto parte de la aplicación anterior para intentar obtener una minimización del tiempo de respuesta de ésta, y para que no haya ciclos de CPU desaprovechados en ninguno de los nodos, hasta que no se concluya la ejecución de la aplicación.

4.2.- Implementación del algoritmo de distribución de carga:

4.2.1.- Introducción:

El algoritmo de distribución de carga, como ya se ha comentado anteriormente, tiene las características de ser distribuido y dinámico.

Al ser distribuido en cada uno de los nodos slaves, existirá una de las unidades que realizan el trabajo de la distribución de carga, y que será igual en cada uno de los nodos.

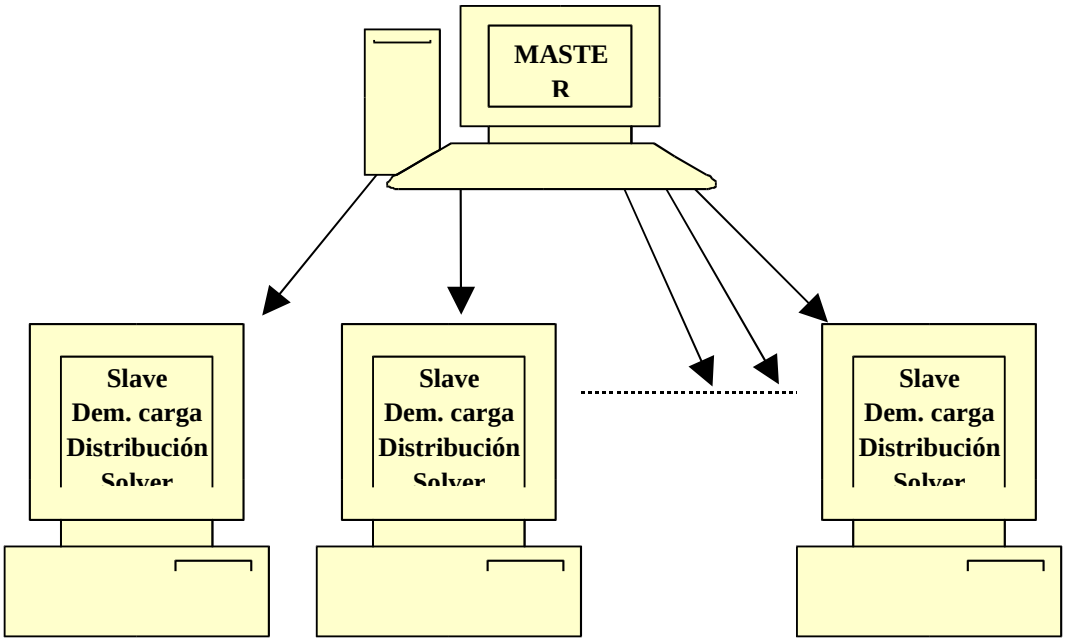
Cada una de las unidades que existen en cada nodo encargada de la distribución de carga, se ha fragmentado en dos procesos:

1.- El primero de ellos se ha denominado **demonio de carga**. La labor fundamental de este proceso es la labor de información sobre el estado del nodo local en el que está ejecutando y conseguir información sobre el estado de todos los demás.

2.- El segundo de ellos es el encargado de realizar las **labores de distribución**. Recibe peticiones de los slaves que soliciten carga y se la manda en el caso necesario.

El esquema general de los procesos existentes en cada nodo es el siguiente:

1.- Un proceso master en un nodo.
2.- Un proceso slave en cada uno de los nodos que realizan el trabajo.
3.- Un proceso de distribución en cada uno de los nodos de los slaves.
4.- Un demonio de carga en cada uno de los nodos de los slaves.
5.- Un proceso solver.



4.2.2.- Grupos de procesos:

La implementación se desarrolla sobre una librería que permite el paso de mensajes (MPI). Por lo tanto todo el desarrollo del algoritmo se basa en la **comunicación entre** los distintos **procesos** mediante mensajes[10, 11].

Para la comunicación de dichos procesos se ha utilizado una de las características que proporciona MPI, y es el hecho de crear **comunicadores distintos** para cada tipo de procesos. Esto significa que se crean distintos grupos para cada uno de los tipos de proceso existentes.

Todos los procesos están incluidos en un grupo principal denominado MPI_COMM_WORLD, mediante el cual cualquier proceso del sistema distribuido se puede comunicar con cualquier otro. Pero si todos los mensajes se realizaran mediante este grupo, todas las funciones de comunicación serían globales, y de este

otro modo se utilizan funciones locales dentro de cada uno de los grupos. Por ello cada tipo de proceso tiene su propio grupo:

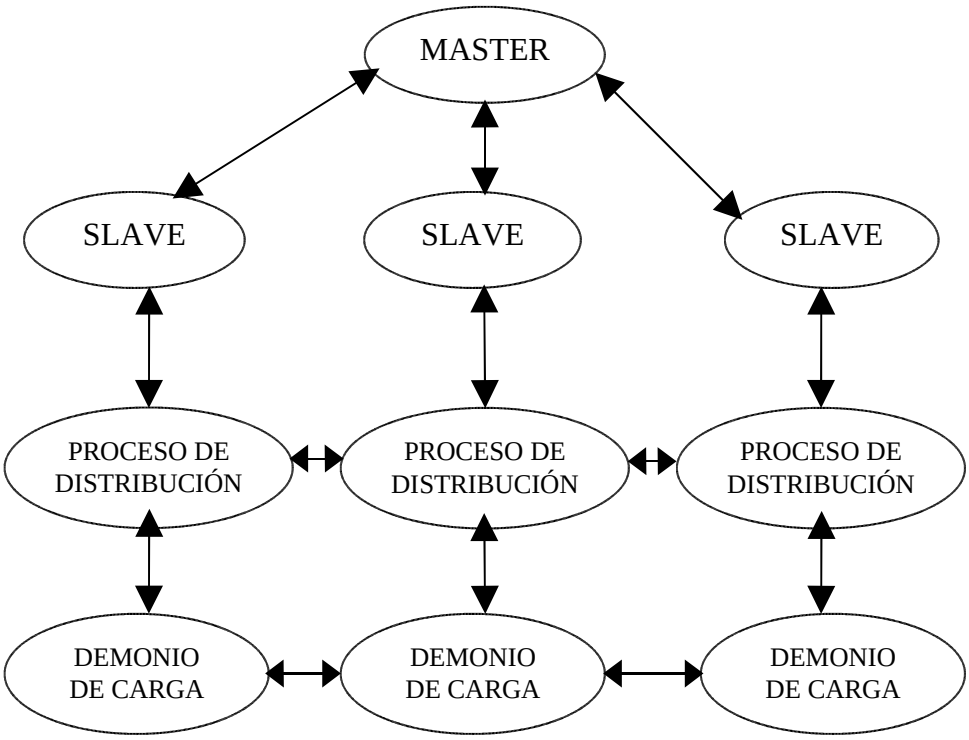
- 1er grupo: *MPI_COMM_WORLD*: es el grupo principal en el que están incluidos todos los procesos.
- 2º grupo: *MPI_COMM_MS*: en este grupo están incluidos el proceso master y todos los procesos slaves.
- 3er grupo: *MPI_COMM_DIST*: a este grupo pertenecen todos los procesos encargados de la distribución (1 por nodo).
- 4º grupo: *MPI_COMM_CARGA*: a este grupo pertenecen los demonios de carga de cada uno de los nodos.

Esta solución evita el continuo paso de mensajes por un mismo comunicador para todos los nodos. De esta forma, cada proceso se comunica con los otros procesos equivalentes en los demás nodos mediante su grupo, y solo utiliza el grupo principal cuando tiene que comunicarse con uno proceso de otro grupo.

El concepto de grupos es la forma más lógica de realizar el esquema de procesos, ya que la mayoría de los mensajes que se intercambian en el sistema se realizan entre procesos del mismo grupo, es decir, los demonios de carga se comunican continuamente entre sí, y solo cuando hay alguna petición se comunican con los procesos de distribución. E igualmente ocurre con los otros procesos, las comunicaciones se realizan principalmente entre procesos del mismo grupo.

Como ya se ha dicho, la mayoría de las comunicaciones se realizan sobre el mismo grupo, pero también hay comunicación entre procesos de distintos grupos. Pero no entre cualquier tipo de procesos, es decir, se mantiene una orden lógico de tipo de procesos para que el paso de mensajes se realice de una forma clara, jerárquica y estructurada.

La **jerarquía** establecida para la comunicación entre procesos de distintos grupos es la siguiente:



Esta jerarquía permite distribuir los mensajes de una forma ordenada, y evitar cuellos de botella que se producirían por ejemplo si todos los procesos se pudieran comunicar con el mismo, por ejemplo con el master. También permite estructurar el sistema, de forma que los mensajes no puedan enviarse de un proceso a cualquier otro, y favorece a que la aplicación sea más estructurada.

4.2.3.- Comunicación mediante mensajes:

La comunicación entre los distintos procesos se realiza mediante el paso de mensajes.

Las primitivas básicas utilizadas son las que permiten mandar y recibir mensajes. Pero esto se puede hacer de varias maneras, dependiendo de la política usada:

1. **Primitivas bloqueantes**: cuando se utiliza este tipo de política, después de utilizar una de las primitivas de comunicación, el flujo de control del programa no continua hasta que el proceso de comunicación ha terminado. En MPI estas operaciones se realizan con: MPI_Recv y MPI_Send.
2. **Primitivas no bloqueantes**: permiten continuar la ejecución del proceso aunque la operación de comunicación no se haya completado. En MPI se realiza mediante MPI_Irecv y MPI_Isend.

En la mayoría de los casos se han utilizado primitivas bloqueantes. Pero se ha utilizado otra utilidad de MPI [10], que permite utilizar una instrucción no bloqueante para comprobar si se ha recibido o no un mensaje, pero el mensaje no se recibe hasta que se ejecuta una de las primitivas bloqueantes señaladas anteriormente. Con esto conseguimos una funcionalidad casi equivalente al uso de primitivas de comunicación no bloqueantes.

Los mensajes constan de varios parámetros:

- Dato que se manda en el mensaje.
- Tipo de dato que se manda.
- Número de datos que incluye el mensaje.
- Nodo destino.
- **Tag**.
- Comunicador.

De todos éstos, el que hay que destacar es el tag. Este parámetro tag no tiene una funcionalidad asignada, y se le puede dar el uso que se quiera. Aquí se ha definido una especie de **protocolo** que utiliza ese parámetro tag, para indicar qué tipo de mensaje se manda.

El esquema básico de los dos procesos desarrollados es el siguiente. Cada uno de ellos puede estar realizando labores específicas, como por ejemplo el demonio de carga, que periódicamente informa del estado del nodo. Pero en todo momento los procesos están esperando recibir algún mensaje. Una vez que lo han recibido, extraen el parámetro tag, y a partir de ese parámetro realizan la función asociada a ese valor.

Esto lo hacen hasta que se cumpla una determinada condición. Esta condición es que no haya carga disponible en ninguno de los nodos para ser transferida. En ese momento esos procesos cambian de estado y pasan a otro estado en el que lo único que hacen es esperar un mensaje que les indique que tienen que terminar su ejecución.

Esquema de ambos procesos:

Mientras que se cumpla la condición1
begin
Esperan a recibir mensajes
Si reciben mensaje, comprueban valor de tag:
Valor 1: Función asociada al valor 1
Valor 2: Función asociada al valor 2
Valor y: Indica que pasen al estado 2
Condicion1 = false
End
Mientras que se cumpla la condición2
begin
Esperan a recibir mensajes
Si reciben mensaje, comprueban valor de tag:
Valor x: Indica que acaben.
Condición2 = false
Otro valor: No hacer nada
end

Los **tipos de mensajes** que pueden recibir los procesos, dependiendo del tag, son los siguientes:

PROCESO DE DISTRIBUCIÓN:

0: Slave indica que ha terminado de procesar.

1: Otro proceso de distribución pide carga.

2: Otro p. de distribución informa nº de firmas que se van a transferir

3: Otro p. de distribución manda firmas

4: Demonio de carga indica número de nodo para transferencia

5: Otro p. de distribución manda tamaño de las firmas que se transfieren

DEMONIO DE CARGA:

0: Información sobre tareas

1: P. de distribución ha terminado y pide nº de nodo para pedir trabajo.

2: P. de distribución informa nº de firmas que ha pasado a otro nodo.

3: P. de distribución indica que no hay nodos para transferir carga.

4: P. de distribución indica nº de firmas que ha obtenido de otro nodo.

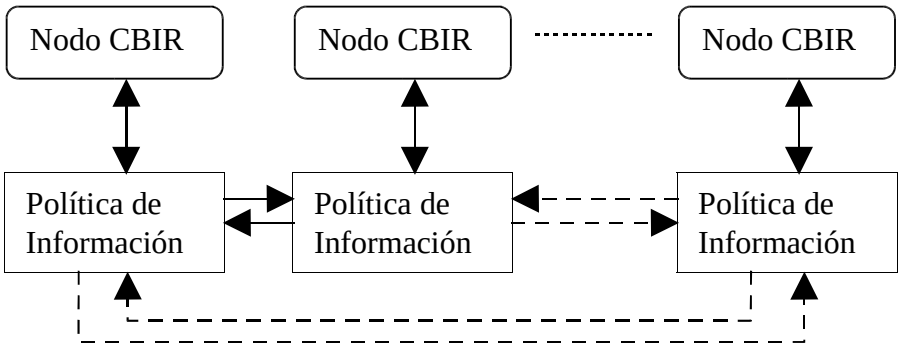
5: Otro demonio de carga informa de su nº de firmas por paso de carga a otro.

6: Otro demonio de carga informa nuevo nº de firmas que ha obtenido.

7: Otro demonio de carga informa que no hay nodos para transferir carga.

4.2.4.- Descripción detallada: Demonio de carga:

La función principal de este proceso, es la de **calcular la carga local**, enviársela a los demás demonios de carga, y enviarle toda la información que tenga al proceso de distribución cuando éste se la pida.



Este proceso tiene una **tabla** en la que va guardando todos los datos de estado sobre los demás nodos. Cada uno de los registros de esta tabla se define así:

```
Struct reg {
        int nodo;
        float velocini;
        float tareas;
        float porcentaje;
        time_t tultact;
        int ttotal;
        long nfirmas;
}
```

El campo nodo contiene el número de nodo en del sistema.

El campo velocini es la velocidad inicial que se ha calculado de cada uno de los nodos. Es la capacidad de proceso de los nodos en vacío.

El campo tareas contiene el número medio de tareas en ejecución durante el tiempo que dure la aplicación.

El campo porcentaje calcula el porcentaje realizado por el solver hasta el momento. Se va calculando dependiendo del número de tareas, la velocidad inicial, los tiempos de actualización y el número total de firmas.

Los campos tultact y ttotal, indican el tiempo de la última vez que se actualizó el valor del registro y el tiempo total de ejecución con una determinada carga respectivamente.

El campo nfirmas indica el número total de firmas (imágenes) que debe procesar ese nodo.

Este proceso está recibiendo mensajes de los demás demonios de carga y de su proceso de distribución hasta que se recibe un mensaje indicando que no hay procesos para transferir carga. En ese momento no finaliza su ejecución, porque otros nodos intentarán comunicarse con él. Entonces, desde ese momento espera a recibir un mensaje del master para finalizar su ejecución, que será el momento en el que todos han finalizado.

El esquema general del proceso es el siguiente:

```
Inicializar_MPI;
Total_firmas = Numero_Firmas;
Inicializar_Tabla;
Tiempoult = time (NULL);
Terminar = 0;
While (!terminar) {
        If (time(NULL)> tiempoult + TIEMPOINF) {
                Enviar_Tareas;
                Actualizar_Tabla;
        } Else {
                Si recibe mensaje de su grupo (MPI_COMM_CARGA):
                        Tag 0: Actualizar_Tabla;
                        Tag 5: Actualizar_Nfirmas;
                        Tag 6: Actualizar_Nuevo_Nfirmas;
                        Tag 7: terminar = 1;
                Si recibe mensaje de COMM_WORLD:
                        Tag 1: Ordenar_Tabla;
                        Tag 2: Informar_Nfirmas;
                                Resto_Nfirmas;
                        Tag 3: Informar_No_Hay_Nodos;
                                Terminar = 1;
                        Tag 4: Nuevo_Nfirmas;
                                Informar_Nuevo_Nfirmas;
        }
}
terminar = 0;
while (!terminar) {
        Si recibe mensaje del master:
                Terminar = 1;
}
```

Este proceso cada cierto periodo de tiempo especificado en TIEMPOINF (definido a priori), envía información de estado a los demás nodos, y continuamente está recibiendo mensajes, que tienen distinto tag, y según el tag que tengan se realizará una labor u otra. A continuación se explican cada una de las funciones desarrolladas en este proceso:

- ***Void Inicializar_MPI (int argc, char **argv)***

  Esta función introduce al proceso en el sistema virtual de MPI, halla el tamaño del comunicador COMM_WORLD, y su rango dentro del comunicador, y el tamaño del mismo. Y de igual manera introduce al proceso en el comunicador MPI_COMM_CARGA, y halla el tamaño y el rango del comunicador.

- ***long Numero_Firmas (char fichero[100])***

  Obtiene el nombre de un fichero de firmas (direcciones de las imágenes), y halla el número de firmas que tiene ese *fichero*, que sería la cantidad de trabajo a procesar.

- ***void Inicializar_Tabla (time_t tult)***

  Lo primero que se hace en esta función es hallar el número de tareas del nodo, y la velocidad inicial.

  Después se manda a todos los demás procesos de carga esta información, la velocidad inicial y el número de tareas.

  Usa las funciones:

  Numero_Tareas;

  Mi_Velocidad;

Y después se espera a recibir todos estos datos de todos los demás nodos y los introduce en la tabla.

- ***int Numero_Tareas (void)***

  Esta función obtiene el número de tareas en ejecución que hay en el nodo en el que se encuentra. Este dato lo obtiene del fichero:

  /proc/loadavg

- ***float Mi_Velocidad (void)***

  Obtiene la velocidad inicial del nodo, que se encuentra almacenada en un fichero del directorio de ejecución, ya que este dato se ha calculado a priori.

- ***void Enviar_Tareas (void)***

  Comprueba el número de tareas local y se lo manda a todos los demás demonios de carga.

- ***void Actualizar_Tabla (int nodo, int ntar, time_t tult)***

  Esta función recibe el número de *nodo*, el número de *tareas*, y el tiempo en que se han recibido estos datos.

  A partir de ahí, el proceso halla el nuevo número medio de tareas, el nuevo porcentaje, y el nuevo tiempo de última actualización de los datos de ese nodo, y con eso actualiza los valores del nodo en la tabla.

- ***void Actualizar_Nfirmas (int nodo, long firm)***

  Actualiza los datos de un *nodo* en la tabla. Esta actualización se debe a que ese nodo le ha pasado carga a otro, y por lo tanto tiene menor número de firmas, indicado en la variable *firm*.

- ***void Actualizar_Nuevo_Nfirmas (int nodo, long firm)***

  Actualiza los datos de un nodo en la tabla. El nodo indicado ha recibido carga de otro nodo, y por tanto hay que actualizar su nuevo número de firmas, sería *firm*, el tiempo que lleva (1) y el porcentaje realizado (0).

- ***void Ordenar_Tabla (void)***

  Esta función es una de las más complejas del proceso.

  Consta de varias partes. En la primera lo que hace es ordenar la tabla de estados según el número de firmas que le quedan por procesar a cada nodo. Para ello utiliza los datos de estado almacenados en dicha tabla.

  Después de esto va seleccionando los nodos que tienen más carga sin procesar, pero de la forma que se comentó en el diseño. Es decir, una vez que se ha ordenado la lista, se obtiene el nodo simétrico al nodo actual en la tabla. Ese sería el primer nodo seleccionado, y si hace falta seleccionar más nodos, se van seleccionando aquellos que se encuentran en la lista con más carga que éste, y cuando se llega al final de la lista, se seleccionan aquellos que se encuentran en la lista con menos carga que éste.

  Cada vez que se selecciona un nodo, se le envía al proceso de distribución, y si éste pide otro se vuelve a seleccionar y a enviarle otro.

  El resultado es que se seleccionan todos, por orden de carga restante, si el proceso de distribución sigue pidiendo nodos, hasta que se recorre toda la lista, que se le manda un código de error.

- ***void Informar_Nfirmas (long firm)***

  Modifica el número de firmas total del nodo actual en la tabla, porque ha pasado carga a otro, y tiene *firm* firmas menos.

- ***long Resto_Nfirmas (long firm)***

  El nodo actual informa a los demás demonios de carga su nuevo nº de firmas, consecuencia de haber pasado carga a otro.

- ***void Informar_No_Hay_Nodos (void)***

  Informa a los demás nodos que no efectúen tareas de búsqueda de nodos para transferir carga porque no hay ninguno que cumpla los requisitos. Este proceso lo sabe porque se lo ha dicho su proceso de distribución, que ha buscado en todos los nodos y no hay ninguno disponible.

- ***void Nuevo_Nfirmas (long firm)***

  Asigna un nuevo número de firmas al nodo actual, ya que ha conseguido carga de otro nodo.

- ***void Informar_Nuevo_Nfirmas (long firm)***

  Informa a los demás demonios de carga que tiene un nuevo número de firmas, ya que ha obtenido carga de otro nodo.

4.2.5.- Descripción detallada: proceso de distribución:

La función principal de este proceso es **recibir peticiones de carga del slave**. Cuando un slave termina, se lo indica al proceso de distribución, y la labor de éste es buscar un nodo adecuado, si lo hay, y entregarle la carga de trabajo adecuada al slave, si es posible.

Es decir, las operaciones que debe realizar este proceso se pueden separar en 2 partes. La primera que es cuando el proceso busca otro nodo para que le transmita carga, y la segunda es cuando el proceso recibe una petición de carga de otro proceso de distribución.

El esquema general del proceso es el siguiente:

```
    Inicializar_MPI;
    Total_firmas = Numero_Firmas;
    Terminar = 0;
    While (!terminar) {
        Si recibe algún mensaje:
            Tag 0: terminar = Pedir_Carga;
            Tag 1: Mandar_Carga;
}
terminar = 0;
while (!terminar) {
        Si recibe mensaje del master:
            Terminar = 1;
`
```

Las funciones incluidas en este proceso son las siguientes:

- ***void Inicializar_MPI (void)***

  Esta función introduce al proceso en el sistema virtual de MPI, halla el tamaño del comunicador COMM_WORLD, y su rango dentro del comunicador. Y de igual manera introduce al proceso en el comunicador MPI_COMM_CARGA, y halla el tamaño y el rango del comunicador.

- ***long Número_firmas (fichero[100])***

  Obtiene el número de firmas que hay en *fichero*.

- ***float Numero_Tareas_Medio (void)***

  Obtiene el número de tareas medio del nodo actual. Este dato lo obtiene del fichero:

  /proc/loadavg

- ***int Pedir_Carga (void)***

  Esta función tiene dos partes:

  1. Primero seleccionar un nodo adecuado para la transferencia. Para ello, se manda al mensaje al demonio de carga, y nos devuelve el nº de nodo. Después le mandamos un mensaje al proceso de distribución de este nodo para preguntarle si tiene carga para transferir. En caso afirmativo se pasa a la segunda parte. En caso negativo se repite esta operación tantas veces como nodos hay en el sistema hasta que se encuentra un nodo adecuado. Si después de recorrer todos los nodos no se encuentra ninguno disponible, se termina el procedimiento y se devuelve un 1. Se envían mensajes al slave y al demonio de carga para indicarles que no hay nodos adecuados para realizar una transferencia.

  2. Una vez seleccionado el nodo, se recibe la carga solicitada, se escribe en un fichero para que la ejecute posteriormente el solver, y se le manda un mensaje al slave y al demonio de carga para indicarles que hay un nuevo trabajo. En este caso el valor devuelto es un 0.

- ***void Mandar_Carga (int nododest, int veloc)***

  Cuando el proceso de distribución entra en esta función es porque ha recibido una petición de carga de otro proceso de distribución (nododest) cuya velocidad inicial es veloc.

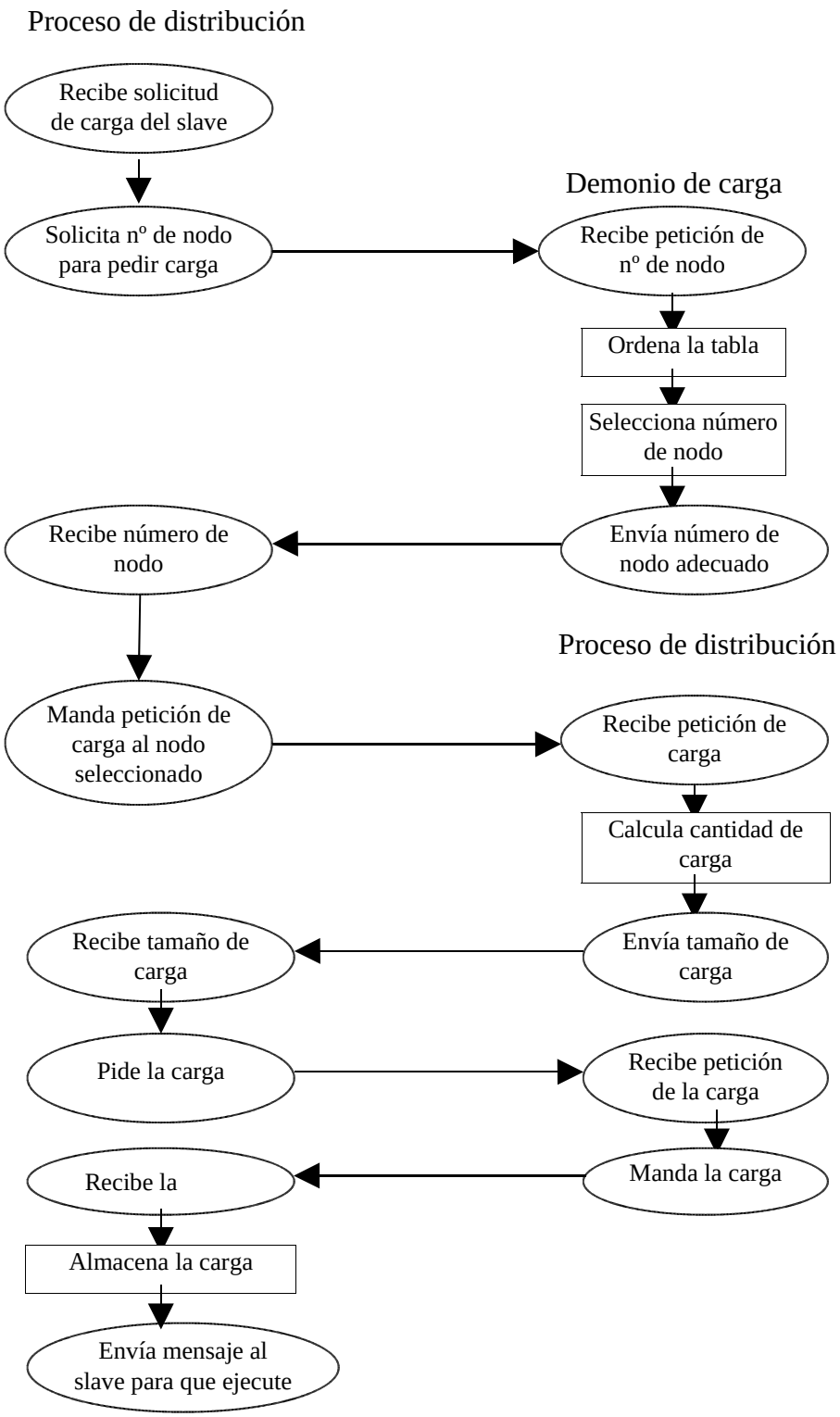
  Lo primero que hace es calcular el número de firmas que puede pasarle al nodo. Para ello comprueba su velocidad inicial y la del otro nodo, su número de tareas y la del otro, y calcula el porcentaje para que ambos procesos acabaran de ejecutar al mismo tiempo. Si este cálculo es mayor que un umbral definido, entonces se le envía un mensaje al otro proceso de distribución indicándole el número de firmas, en caso contrario se le envía el mismo mensaje con número de firmas 0.

  Si el número que hemos enviado es mayor de 0, pasamos a enviarle el tamaño de lo que le vamos a enviar, y después le pasamos la carga.

  Después de esto informamos al demonio de carga de la reducción en el número de firmas que hay en el nodo.

4.2.6.- Esquema de funcionamiento del algoritmo:

A continuación se presenta un esquema de todos los pasos que se llevan a cabo cuando un slave termina. Le pide carga al proceso de distribución, éste la consigue y se la entrega para que siga ejecutando:


Proceso de distribución
Recibe solicitud de carga del slave
Solicita nº de nodo para pedir carga
Demonio de carga
Recibe petición de nº de nodo
Ordena la tabla
Selecciona número de nodo
Envía número de nodo adecuado
Recibe número de nodo
Proceso de distribución
Manda petición de carga al nodo seleccionado
Recibe petición de carga
Calcula cantidad de carga
Envía tamaño de carga
Recibe tamaño de carga
Pide la carga
Recibe petición de la carga
Manda la carga
Recibe la
Almacena la carga
Envía mensaje al slave para que ejecute

RESULTADOS:

5.1.- Introducción:

A continuación se presentan los resultados obtenidos de la elaboración del algoritmo de distribución de carga desarrollado en el presente proyecto. Dichos resultados se comparan con los obtenidos de la aplicación sin realizar una distribución de carga.

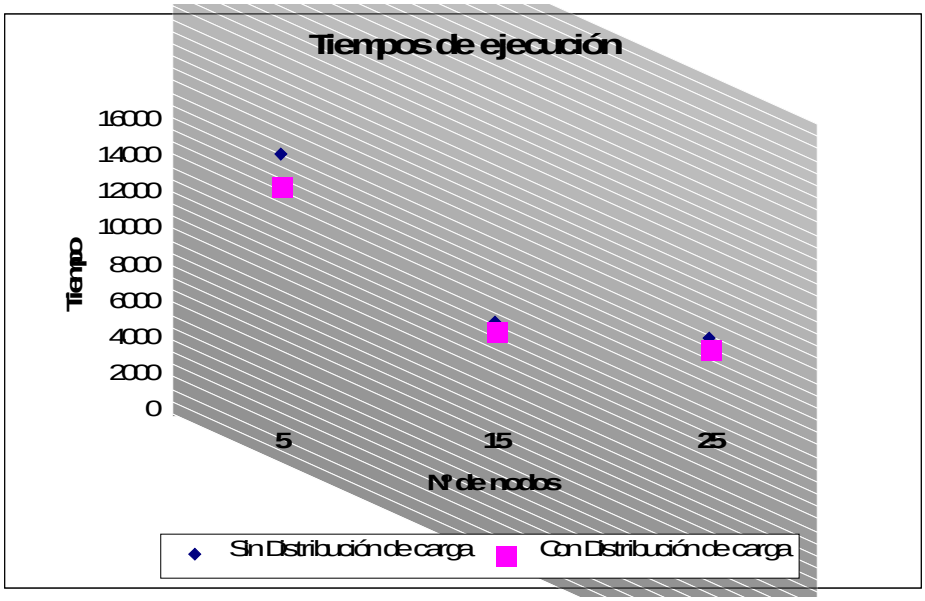
Las pruebas se han realizado sobre un cluster de 26 PCs heterogéneos. 25 de ellos se usan como slaves y uno de ellos como master.

Se han separado en dos partes. La primera se presentan los resultados con el sistema muy poco cargado, y ninguno de los nodos sobrecargados. En este caso se ha usado una base de datos de 30 millones de imágenes.

Y después se presentan los resultados obtenidos con el sistema poco cargado pero uno de los nodos sobrecargados. Para realizar esta segunda parte se ha desarrollado un pequeño programa para introducir carga adicional a uno de los nodos. Y una base de datos de 12.5 millones de imágenes.

5.2.- Resultados sin carga local:

5.2.1.- Tiempo de ejecución:

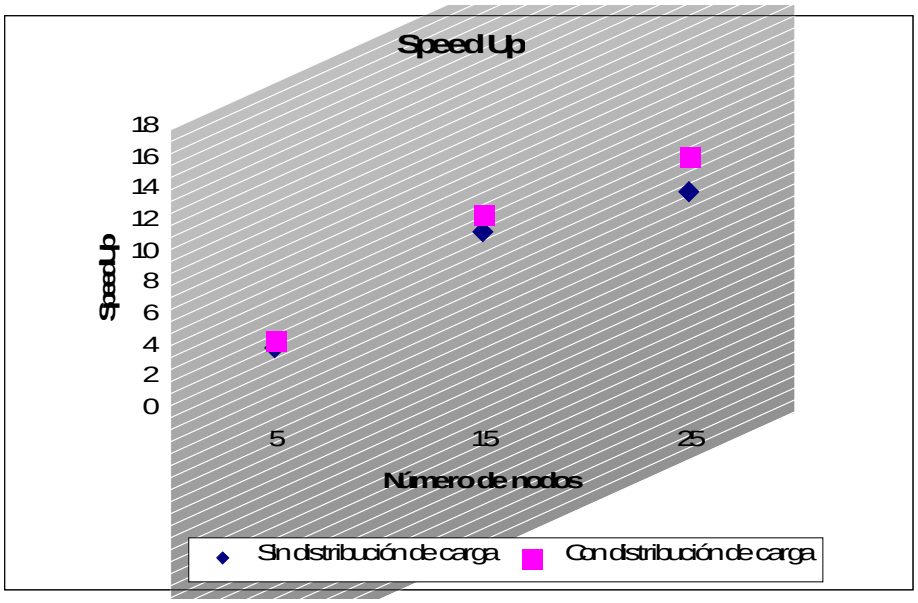


	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	14362	5110	4169
Con distribución de carga	12628	4663	3593

En este gráfico se comprueba el tiempo de ejecución de la aplicación frente al número de nodos usados. Se realizan pruebas con 5, 15 y 25 nodos. Y se puede observar que utilizando el algoritmo de distribución de carga el tiempo de ejecución se reduce en un 12 %, 8.7 % y un 13.8 % respectivamente para cada una de las pruebas desarrolladas.

Por lo tanto se ha reducido el tiempo de ejecución de la aplicación en todas las pruebas, y se reduce más cuantos más nodos hay en el sistema utilizando el algoritmo de distribución.

5.2.2.- Speed Up:



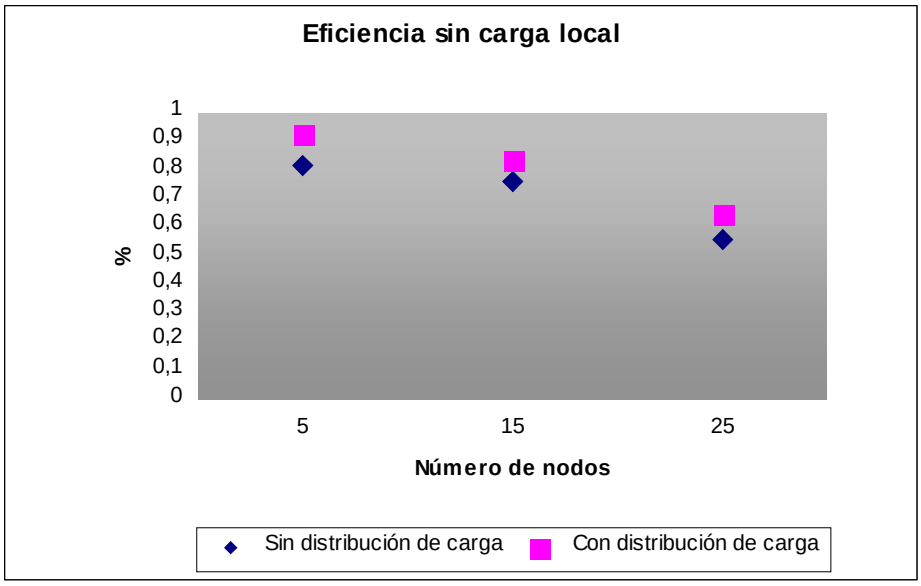
	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	4.084	11.479	14.07
Con distribución de carga	4.645	12.579	16.325

En este gráfico se muestra el Speed Up del sistema con cada una de las pruebas. El Speed Up se define como el tiempo de ejecución en 1 nodo, dividido por el tiempo de ejecución en N nodos.

El Speed Up ideal sería N. Con estos resultados se muestra que el Speed Up es más alto con el uso del algoritmo que sin usarlo, lo que indica un buen aprovechamiento del sistema.

Se observa que cuantos más nodos hay en el sistema, la línea de Speed Up se va alejando más de la línea ideal, que sería de 45º, pero vemos que se acerca más que las pruebas sin distribución de carga.

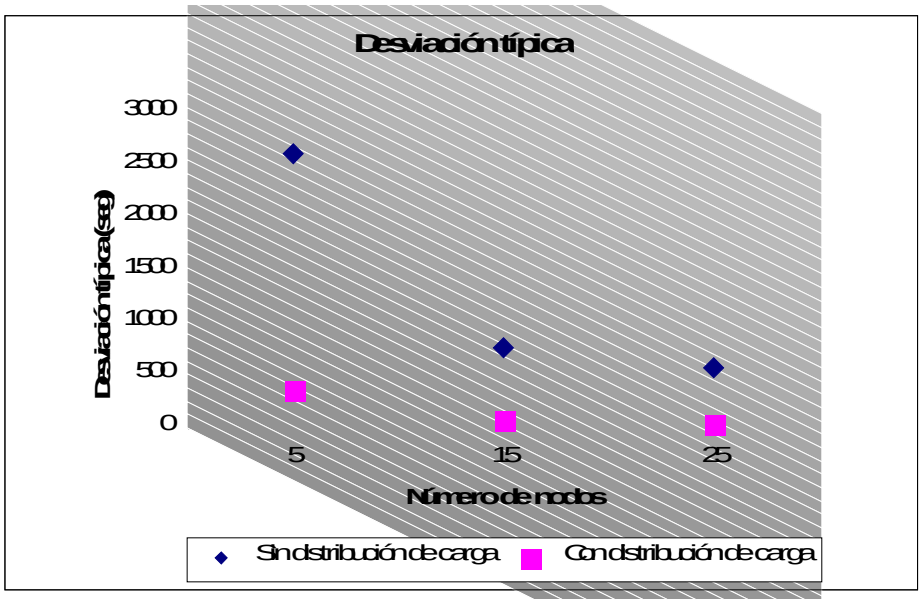
5.2.3.- Eficiencia:



La eficiencia se mide dividiendo el Speed Up dividido del número de nodos del sistema. Observamos que la línea va alejándose del 1, que sería la línea ideal de eficiencia.

Pero al igual que en los otros casos la eficiencia obtenida con el algoritmo de distribución mejora la obtenida sin usar dicho algoritmo. Y se acerca más a esa línea ideal.

5.2.4.- Desviación Típica:



	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	2620	763	575
Con distribución de carga	355	73	37

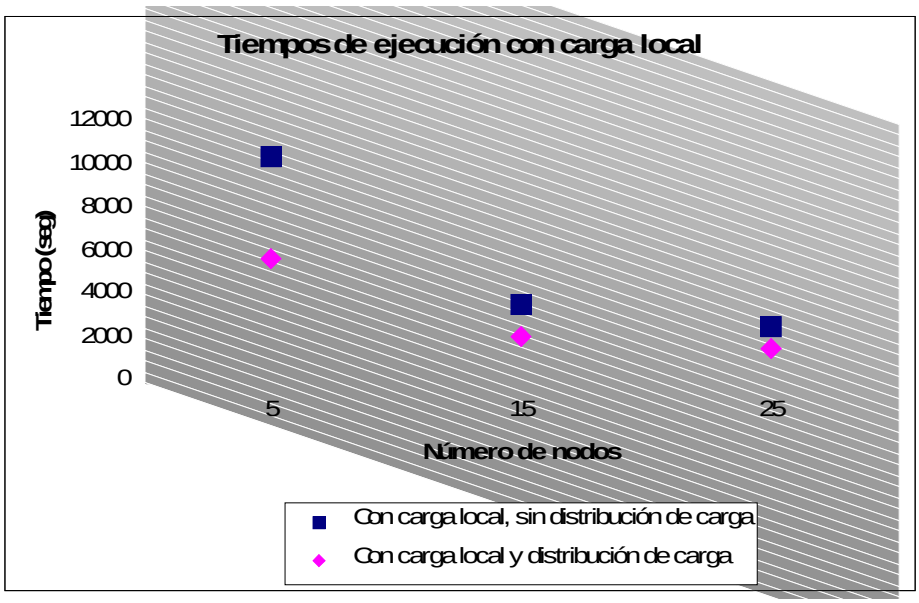
En este gráfico se muestra la desviación típica de la aplicación tomando los valores de tiempo de ejecución de cada uno de los nodos.

Observamos la gran diferencia entre la desviación típica usando el algoritmo y sin usarlo. Esto es debido a que cuando no se usa el algoritmo cada nodo termina en un determinado tiempo. Pero cuando se usa el algoritmo se consigue que todos los nodos terminen “más o menos” al mismo tiempo. Con lo que se consigue una reducción drástica de la desviación típica.

Éste era uno de los principales objetivos del desarrollo del algoritmo, ya que de este modo se asegura que todos los nodos terminan al “mismo” tiempo, y por tanto no se desperdician ciclos de CPU en un nodo que podrían utilizarse en otro.

5.3.- Resultados con carga local:

5.3.1.- Tiempos de ejecución con carga local:

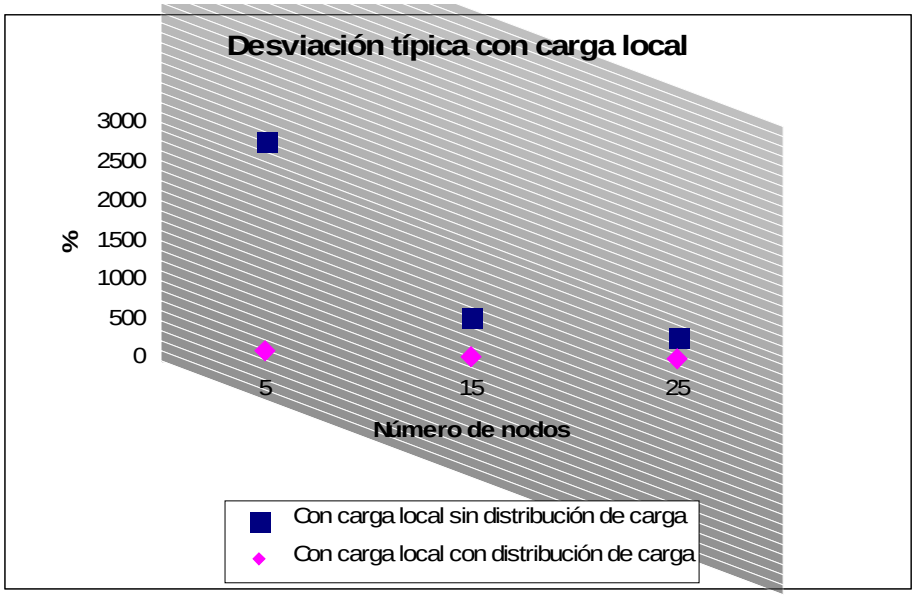


	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	10548	3645	2589
Con distribución de carga	5732	2099	1601

Estos resultados muestran la gran diferencia de tiempo de ejecución entre los tiempos obtenidos con carga y con distribución, y los tiempos con carga y sin distribución. La reducción de tiempo es de un 45% en la primera prueba, un 42% en la segunda y un 38% en la tercera.

Es en estos casos en los que existe carga local en alguno de los nodos, cuando la eficiencia del algoritmo aumenta realmente. Con el algoritmo desarrollado se utilizan los ciclos de CPU que se desaprovechan cuando no existe distribución. Por eso la diferencia tan grande de tiempos entre un usar el algoritmo y no usarlo.

5.3.2.- Desviación típica con carga local:



	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	0.816	0.765	0.56
Con distribución de carga	0.929	0.836	0.653

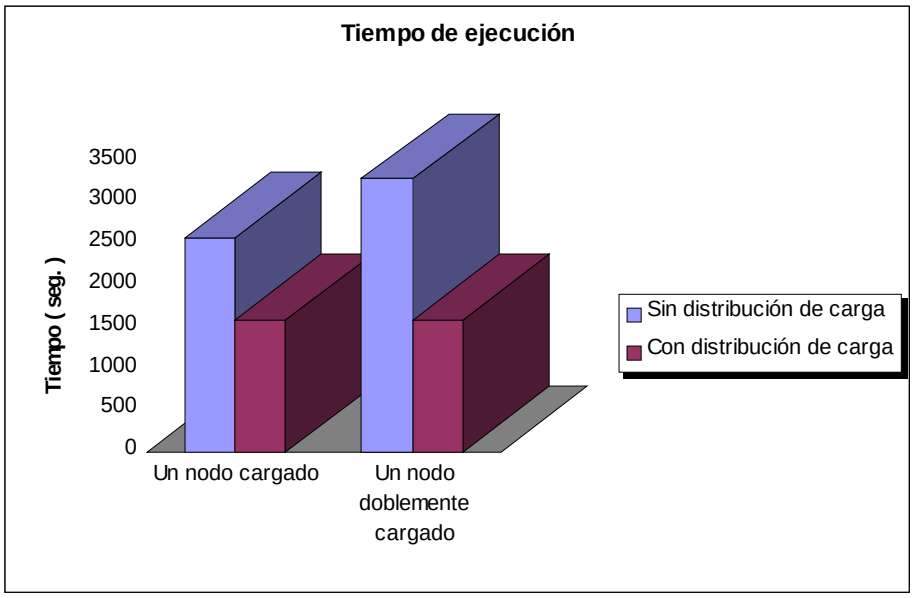
	5 Nodos	15 Nodos	25 Nodos
Sin distribución de carga	2809	565	308
Con distribución de carga	127	54	16

Y si los resultados anteriores eran muy buenos, éstos no lo son menos. La desviación típica disminuye tanto, que se aproxima mucho al 0, que es el objetivo. En el primer caso se reduce hasta el 4%, en el segundo se reduce al 9% y en el tercer caso disminuye hasta el 5%.

Estos datos vuelven a reafirmar que el algoritmo proporciona los mejores resultados cuando hay algo de carga en el sistema, y que aprovecha al máximo todos los nodos, ya que hace que terminen al “mismo” tiempo de ejecutar todos ellos.

5.3.3.- Comparativa del sistema con un nodo cargado:

A continuación se muestran unos resultados obtenidos introduciendo carga local en uno de los nodos. Para comprobar como varia el sistema cuando se introduce más carga, se han realizado pruebas con 25 nodos, y se ha añadido carga a uno de ellos, pero en una prueba se ha introducido el doble de carga que en la otra.



	Nodo cargado	Nodo doblemente cargado
Sin distribución de carga	2589	3293
Con distribución de carga	1601	1601

Se observa en el gráfico que los tiempos con distribución de carga, es decir, usando el algoritmo son exactamente iguales cuando un nodo está doblemente cargado que en el primer caso. Estoy es así gracias a que los demás nodos terminan antes que éste, y procesan la carga que tiene este nodo. Por eso los tiempos son iguales.

En cambio, sin utilizar distribución, los tiempos aumentan un 27% más. Y esto demuestra que mientras haya nodos que no están sobrecargados, la reducción del tiempo de ejecución usando el algoritmo es muy grande.

CONCLUSIONES:

- Para el diseño del algoritmo se han tenido que estudiar los sistemas distribuidos, se ha estudiado su funcionamiento y la variación del rendimiento de estos sistemas, cuando el número de nodos del sistema varía. Y se puede deducir que estos sistemas son muy eficientes cuando el algoritmo es adecuado.

- Se ha hecho un estudio de los diferentes algoritmos de distribución de carga existentes. Se han estudiado las ventajas y los inconvenientes de cada uno de ellos, se han estudiado cada una de las características de cada uno de éstos algoritmos, y se han extraído las características que se ha creído que eran más adecuadas para su posterior desarrollo.

- A partir de estas características extraídas del estudio de los diferentes algoritmos de distribución se ha diseñado un algoritmo con las opciones que se han creído más eficientes. Dos de las más importantes han sido que el algoritmo sea distribuido y dinámico. Dos características que influyen en gran medida en el desarrollo global de la aplicación desarrollada, que proporcionan una gran escalabilidad y evitan problemas tales como cuellos de botella y sobrecarga de un nodo central.

- Se ha estudiado una aplicación de tipo CBIR. Estas aplicaciones necesitan una gran capacidad de cómputo, debido al gran conjunto de datos que necesitan gestionar. Y por ello este tipo de aplicaciones son muy adecuadas para realizar una distribución de las mismas.

- Para el desarrollo del algoritmo se ha utilizado MPI, esta librería permite el paso de mensajes para realizar aplicaciones paralelas. Se ha estudiado dicha librería y se han comprendidos todos los problemas y las ventajas que proporciona la misma. Problemas típicos de aplicaciones paralelas han aparecido en el desarrollo del algoritmo tales como problemas de ínter bloqueos, cuellos de botella, etc ...

- La implementación del algoritmo se ha desarrollado mediante varios procesos. Esta elección se ha hecho para diferenciar bien las diferentes políticas analizadas en el análisis, y una vez desarrollado el algoritmo se puede afirmar que la elección ha sido la correcta, porque de otro modo se habría sobrecargado de mensajes a algún proceso, y eso influye negativamente en la eficiencia del algoritmo.

Las conclusiones más relevantes se pueden realizar a partir de la obtención de los resultados:

- La distribución de carga es un método efectivo para conseguir mejores resultados de ejecución con aplicaciones distribuidas. Las mejoras obtenidas son de aproximadamente un 12 % sin carga

local, y cuando hay carga local en un nodo las mejoras son de un 45 % en tiempo de ejecución.

- Cuando el número de nodos en el sistema aumenta, el porcentaje de la ganancia de tiempo de ejecución aumenta. Es decir, cuantos más nodos hay en el sistema, más ganancia se produce.
- Tanto el Speed Up como la eficiencia del sistema con el algoritmo, se acercan más a las líneas ideales de cada una de las pruebas.
- El aumento de carga en uno de los nodos del sistema, no afecta al sistema en global cuando se realiza distribución de carga.
- De lo anterior, se puede deducir, que mientras que en el sistema haya nodos que no están sobrecargados, se producirán mejoras de tiempo. Ya que los nodos que terminan antes su ejecución realizan el trabajo de los nodos sobrecargados. Por ello la desviación típica disminuye hasta en un 95%.
- Todas las pruebas realizadas con el algoritmo de distribución han sido favorables, por lo tanto, la distribución de carga es un componente muy importante a tener en cuenta en el futuro desarrollo de aplicaciones distribuidas.

BIBLIOGRAFÍA:

[1] ZHOU, S. A trace driven simulation study of dynamic load balancing. (Septiembre 1988).

[2] THEIMER, M., AND LANTZ, K. Finding idle machines in a workstation based distributed system. (Noviembre 1989).

[3] OZDEN, B., GOLDBERG, A., AND SILBERSCHATZ, A. Scalable and non-intrusive load-sharing in owner-based distributed systems. (Diciembre 1993).

[4] KREMIEN, O., AND KRAMER, J. Methodical analysis of adaptive load sharing algorithms. (Noviembre 1992).

[5] LULING, R., MONIEN, B., AND RAMME, F. A study on dynamic load balancing algorithms. (Junio 1992).

[6] SHIN, K. G., AND CHANG, Y.-C. A coordinated location policy for load sharing in hipercube-connected multicomputers. (Mayo 1995).

[7] EAGER, D., LAZOWSKA, E., AND ZAHORJAN, J. Adaptive load sharing in homogeneous distributed systems. (Mayo 1986).

[8] STANKOVIC, J. Stability and distributed scheduling algorithms. (Octubre 1985).

[9] NI, L., XU, C., AND GENDREAU, T. A distributed drafting algorithm for load balancing. (Octubre 1985).

[10] OHIO SUPERCOMPUTER CENTER. MPI Primer/Developing with LAM. (Noviembre 1996).

[11] JOSÉ MIGUEL ALONSO. Programación de aplicaciones paralelas con MPI. (Enero 1997).