Proyecto de Sistemas Informáticos
Curso 2008-2009.

Motores de Búsqueda usando UPC

Componentes del grupo:

- Jorge Olmos Mallol
- Eduardo Hombrados Carrillo
- Rubén Jesús Ruiz Cuadrado

Director del proyecto:

- Manuel Prieto Matías (DACYA)

Facultad de Informática.
Universidad Complutense de Madrid.

Índice general

I Resumenes y Licencia de Uso **1**

0.1. Autorización 3

0.2. Resumen 4

0.2.1. Palabras clave 4

0.3. Abstract 5

0.3.1. Keywords 5

0.4. Agradecimentos 6

II Introducción **7**

1. Buscadores de información **9**

1.1. Introducción 9

1.1.1. Breve historia de la IR 9

1.2. Partes de un buscador 10

1.3. IR Distribuida 12

1.3.1. Aplicación a los prototipos de motor de búsqueda 12

2. Modelo de programación PGAS **15**

2.1. Modelos de programación paralela 15

2.1.1. Modelo con paralelismo de datos 15

2.1.2. Modelo de paso de mensajes 15
2.1.3. Modelo de memoria compartida 18
2.1.4. El modelo PGAS 19
2.2. El modelo de espacios de direcciones global particionado 19
2.2.1. Breve reseña histórica 19
2.2.2. La visión de un programa PGAS 20
2.2.3. La visión de memoria para un programa PGAS 20
2.2.4. Sincronización básica para un programa PGAS 20
2.2.5. Rendimiento para un programa PGAS 21
2.2.6. Lenguajes basados en el modelo PGAS 21

III Implementación y resultados 31

3. Implementacion de un buscador 33

3.1. Operaciones de lectura con escrituras offline 33
3.1.1. Introducción 33
3.1.2. Pruebas preliminares 33
3.1.3. Distribución de los datos 34
3.1.4. Comunicación entre threads 38
3.1.5. Funcionamiento del programa 39
3.1.6. El algoritmo de ranking 41
3.1.7. Manual de uso 42
3.1.8. Pruebas y resultados experimentales 43
3.2. Operaciones de lectura con escrituras online poco frecuentes 48
3.2.1. Introducción 48
3.2.2. Distribución de los datos 48
3.2.3. Operaciones 51

4. Conclusiones **55**

4.1. Sobre la implementación del buscador 55

4.2. Sobre el lenguaje UPC 55

IV Apéndices **57**

Configuración de un cluster virtual **59**

.1. Motivación 59

.2. Manual de configuración 59

Configuración e instalación del entorno de trabajo del cluster virtual. **75**

.2.1. Compartir archivos y aplicaciones 75

.2.2. Instalación del Compilador GCC-UPC 77

.2.3. Instalación del base runtime de Berkeley 78

.2.4. Gestor Sun Grid Engine 80

Bibliografía **83**

Índice de figuras

1.1. Estructura básica de un buscador (Web) 11

2.1. Modelo de paralelismo de datos 16
2.2. Modelo de paso de mensajes 17
2.3. Modelo de paso de mensajes 18
2.4. Modelo de memoria compartida distribuida 23
2.5. Matrices V en NMF 26
2.6. Matrices W en NMF 26
2.7. Matrices H en NMF 27
2.8. Matrices 1 y 2 en NMF 27
2.9. Matriz 3 en NMF 28
2.10. Resultados de NMF para k=8 duplicando el tamaño de problema . . . 28
2.11. Resultados de NMF para k=16 cuadruplicando el tamaño de problema 29

3.1. Modelo ideal de distribución de las posting list vs. distribución adaptada a las limitaciones de los punteros empaqueados 34
3.2. Vision en modelo ideal del reparto de documentos por término 35
3.3. Flujo del programa 39
3.4. Resultados en solo lectura para K=64 45
3.5. Resultados en solo lectura para K=128 46
3.6. Estudio de productividad segun el tamaño de lote 47

3.7. Comparativa de productividad entre versiones con/sin optimizacion . . 48
3.8. Cabecera de bloque de documentos 50
3.9. Bloque de documentos en la versión con escritura 50

1. Crear una nueva máquina virtual 60
2. Selección del tipo de VM 60
3. Selección compatibilidad de hardware de la VM 61
4. Selección guest OS de la VM 61
5. Selección de nombre y ruta de la VM 62
6. Selección del número de procesadores de la VM 63
7. Selección de la RAM disponible para la VM 64
8. Selección de la red usada por la VM 65
9. Selección de interfaz de almacenamiento para VM 65
10. Selección de disco duro para la VM 66
11. Selección de tipo de disco para la VM 67
12. Selección capacidad de disco en una VM 67
13. Selección de ruta para disco virtual 68
14. Vision de la red del cluster virtual 69
15. Menú de configuracion de una VM 69
16. Creacion de un nuevo interfaz de red 70
17. Selección de propiedades del interfaz de red 70
18. El menu de selección de KNetworkManager 71
19. Configurando los adaptadores en linux 71
20. Comprobación de la conectividad en el cluster 72
21. Configuración de los nombres de los equipos 73

22. Panel de coonfiguración de KDE 76

23. Panel de coonfiguración de NFS 76
24. Edicion de fstab 77
25. Resultado de lanzar el comando upc -v 78
26. Resultado de lanzar el comando upc -v 79
27. Resultado de lanzar el comando upcc –version 79

Índice de cuadros

3.1. Resultados para K=64 44
3.2. Resultados para K=128 45
3.3. Resultados de ajuste de tamaño de lote 47
3.4. Resultados para K=128 con optimizaciones 47

Parte I

Resumenes y Licencia de Uso

0.1. Autorización

Autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria, como el código , la documentación y/o el prototipo desarrollado.

Firmado: Jorge Olmos Mallol

Eduardo Hombrados Carrillo

Rubén Jesús Ruiz Cuadrado

0.2. Resumen

En este documento se describen aspectos teóricos y prácticos sobre motores de búsqueda paralelos. El objetivo principal de este trabajo consiste en el diseño e implementación en UPC de un motor de búsqueda para operaciones de tipo OR que sea capaz de obtener un buen redimiento en escenarios de tráfico elevado.

Para ello el documento comienza con una introducción a los sistemas de recuperación de información. Se incluye un breve paseo por la historia de los métodos de clasificación y ordenación de colecciones de documentos por el ser humano, así como una descripción más detallada del modelo general de sistemas de búsqueda por computador. Tambien mencionaremos aspectos sobre recuperación de información de forma distribuida.

La introducción teórica continúa con la presentación de los diversos modelos de programación paralela, centrándonos en el modelo PGAS y en concreto en el lenguaje UPC que ha sido el elegido para la parte práctica del proyecto.

Este documento incluye además información detallada sobre la implementación realizada de un motor de búsqueda centrándose en varios aspectos esenciales: modelos de datos, algoritmos de ranking, comunicación entre nodos y flujo del programa. El análisis del diseño se acompaña de varias pruebas de rendimiento diseñadas para medir diversos parámetros de la implementación.

Finalmente se plantean una serie de conclusiones sobre diversos aspectos encontrados durante el desarrollo del proyecto, así como diversos manuales que contienen la información necesaria para poder replicar los entornos de trabajo utilizados.

0.2.1. Palabras clave

Recuperación de información,PGAS,UPC,motor de búsqueda

0.3. Abstract

This document describes theorical and practical aspects about parallel search engines. The aim of this work consists in the design and implementation in UPC of a search engine for OR-type operations being able to offer good performance in high traffic scenarios.

To accomplish that the document begins with an introduction to information retrieval sistems. It includes a brief walkthrough about the history of methods used by people to classify and order collections of document, and a more detailed description about the general model for computer search systems. It also explains some aspects about distribuited information retrieval.

A presentation of various parallel programming models follows the theorical introduction. We will focus on PGAS model and, specifically, on UPC language which has been our choice to implement the practical part of this project.

This document also includes detailed information about our search engine implementation, focusing on various essential aspects: data models,ranking algorithms, node communication and program flow. The analysis of the design is followed with some performance tests aimed at measure various parameters of the implementation.

Finally it exposes a series of conclusions about various aspects we met during the development of the project and some manuals which contain the necessary information needed to be able to replicate the work environments we used.

0.3.1. Keywords

Information retrieval,PGAS,UPC,search engine

0.4. Agradecimentos

A Carolina Bonacic Castro por facilitarnos la documentación de su Master de Investigación, y por respondernos a tantas y tantas preguntas sobre los algoritmos de vectoriales de ranking que tan raros nos parecían en un principio.

A Manuel Prieto Matías por dirigir el proyecto y por los quebraderos de cabeza que le provocábamos cada vez que nos quedábamos bloqueados ante un problema de especial dificultad.

A Maria Isabel Pita Andreu por las transparencias que han inspirado la introducción histórica contenida en esta memoria y por facilitarnos algunas referencias que nos han sido muy utiles para comprender la estructura general de un motor de búsqueda.

A los técnicos de laboratorio que han tenido que soportar nuestra inexperiencia en Linux y nuestras quejas cuando habia problemas de funcionamiento del cluster y se aproximaban los *deadlines*

Parte II

Introducción

Capítulo 1

Buscadores de información

1.1. Introducción

El proceso de recuperación de información (IR, *information retrieval*) se ocupa de la representación, almacenamiento, organización y acceso a elementos de información. En concreto se refiere al proceso de búsqueda de una información en particular dentro de una colección de documentos.
En contraposición con la recuperación de datos clásica, que busca obtener todos los objetos que satisfacen condiciones claramente definidas (como las de las expresiones regulares), la IR normalmente trata con texto en lenguaje natural que no está siempre bien estructurado y es naturalmente ambiguo. Por ello en un sistema de IR los objetos recuperados pueden ser imprecisos y los pequeños errores normalmente no tienen importancia.

1.1.1. Breve historia de la IR

Antes de la invención del papel los romanos y los griegos almacenaban la información en rollos de papiro. Algunos papiros tenían asociada una etiqueta que contenía un pequeño resumen del documento para facilitar su búsqueda. Es en el siglo segundo A.C cuando aparecen los primeros índices en rollos de papiro griegos. De está epoca se desconocen más detalles (como por ejemplo la clasificación que usaban en las bibliotecas para los papiros) [14]

En la Edad Media las bibliotecas estaban localizadas en los monasterios. Normalmente tenian un tamaño muy pequeño y por tanto no necesitaban ningún sistema de clasificación. Cuando las colecciones empezaron a aumentar de tamaño se empezaron a dividir en tres grupos: trabajos teológicos,autores clásicos y contemporáneos. En esta época se contaba ya con listados completos de las obras e índices de los contenidos.

Con la llegada de la imprenta en 1450 las colecciones de las bibliotecas crecieron y fueron accesibles a todo el público. Se empezaron a utilizar sistemas de clasificación

jerárquicos para agrupar los documentos por materias.

Durante el siglo XX, con el gran aumento de las colecciones de documentos, los bibliotecarios empezaron a usar listados ordenados por diversas claves (título, autor), microfilms (1930), y el sistema MARC (MAchine Readable Cataloging - 1960).

Con la llegada de las primeras computadoras (1940) aparecen los primeros sistemas de búsqueda por computador. Sin embargo en los primeros sistemas la sintaxis era realmente compleja y su uso se limitaba a los bibliotecarios. Podemos considerar tres generaciones de sistemas de IR modernos:

- La primera generación consistía en una automatización de las tecnologías previas (sistemas de tarjetas y similares). Permitía las busquedas por autor y por título.
- En la segunda generación se amplian las funcionalidades de búsqueda con añadidos como poder buscar por asuntos, palabras clave y en general permitiendo consultas más complejas.
- La tercera generación (la actual) se centra en mejorar los interfaces gráficos, capacidades de hipertexto, arquitecturas de sistema abiertas...

La última revolución en el mundo de los buscadores de IR llegó con la aparición de la World Wide Web. Los motivos de esta revolución han sido los siguientes:

- El acceso a las fuentes de información se ha abaratado lo que permite alcanzar una audiencia mayor que nunca.
- Los avances en tecnología han conseguido que se dispare en acceso a las redes de información. Ahora cualquier fuente de información está disponible sin importar la distancia geográfica y de forma rápida.
- La facilidad que tiene el usuario para colgar en la red cualquier información que considere interesante. Esto ha contribuido en gran medida a la expansión de la Web.

1.2. Partes de un buscador

El comportamiento general de un buscador de compone de varias partes diferenciadas. Éstas pueden variar ligeramente en función de las caraccterísticas de los datos o de la implementación, pero se ajustan con bastante exactitud al siguiente modelo:

Un sistema para obtener información. Puede ser un medio humano o un sistema automatizado como es el caso de los *crawlers* en la World Wide Web. Su cometido es recuperar documentos y almacenarlos en algun sistema de bases de datos de acuerdo a un modelo válido para los documentos que se han obtenido.

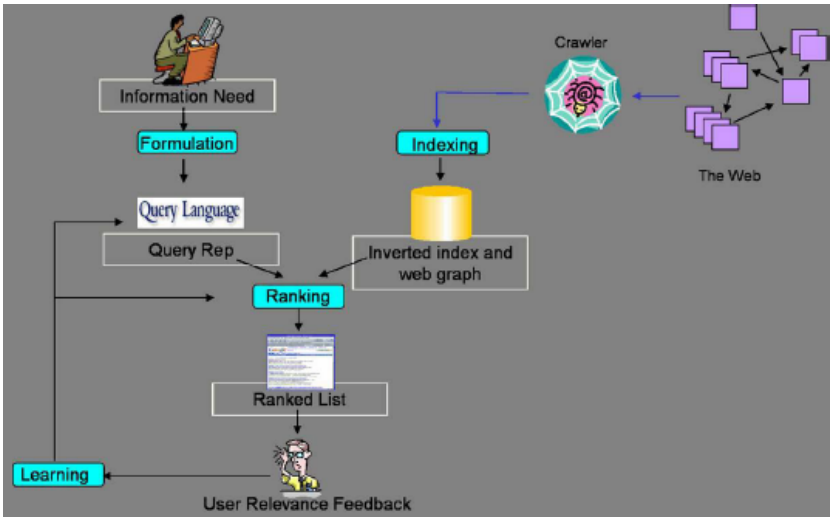

Figura 1.1: Estructura básica de un buscador (Web)

Un módulo para generar índices. Dicho módulo analiza los documentos extrayendo determinada información clave que se usará para poder buscar de forma rápida sobre todo el volumen de datos almacenados. Aunque los índices se pueden implementar con diversas estructuras, la más común es la de *lista invertida*. Aunque el proceso de indexado es muy costoso, en un sistema real este coste se amortiza de forma rápida con la mejora de rendimiento en las consultas.

Un sistema de consultas. Usando este sistema el usuario puede lanzar consultas para recuperar información del sistema. Las consultas suelen ser preprocesadas de la misma manera que lo fueron los textos originales antes de ser indexados. El sistema de consultas se encarga de aplicar la consulta preprocesada contra la base de datos (usando los índices invertidos generados por el módulo de indexado) y obtiene los documentos que se ajustan a la consulta.

Un módulo de puntuación (Ranking) que puntúa los documentos recuperados de acuerdo a criterios de relevancia respecto a la consulta del usuario. Existen numerosos algoritmos de ranking, desde los booleanos mas simples, pasando por los vectoriales o los de lógica difusa.

En algunos casos hay un sistema de feedbackcon el cual los usuarios refinan la consulta (y con suerte ésta es una representación mejor de los deseos del usuario). El

feedback se puede aprovechar para modificar las puntuaciones asociadas a documentos en el sistema de ranking.

1.3. IR Distribuida

El aumento de tamaño de las colecciones de documentos y del número de peticiones que se realizan contra éstas hace casi imposible que un sistema de búsqueda real se asiente sobre una única máquina. Por ello el concepto de IR distribuida ha tomado especial relevancia en los últimos años.
Podemos considerar la IR Distribuida en cierta manera como si se tratara de IR Paralela en máquinas MIMD que tienen una red de interconexión lenta. Cada una de las máquinas luego puede soportar algún tipo de procesamiento paralelo de forma individual. Por eso interesa que la comunicación entre máquinas sea la menor posible.
Cuando se habla de IR Distribuida entra en funcionamiento una parte adicional del motor de búsqueda: *el broker*. El broker es una máquina (o conjunto de ellas) que se encarga de, una vez recibida una *query*, reenviarla a los ranker que estime oportunos buscando mantener un equilibrio de carga entre ellos. Este envio puede ser por broadcast o bien seleccionando rankers individuales de acuerdo a alguna heurística.
El proceso por tanto de IR Distribuida puede verse de la siguiente manera:

1. Llega una consulta al sistema.
2. El broker selecciona los nodos que van a encargarse de la query.
3. El broker envia la query a los nodos seleccionados.
4. Se evalua la query en paralelo.
5. Se combinan resultados.

1.3.1. Aplicación a los prototipos de motor de búsqueda

En el prototipo de motor de búsqueda que se desarrollará el capítulo 3.1 se simula un broker mediante un archivo de consultas. El ranker encargado de una cierta consulta extraerá datos de los otros nodos y con ellos generará una evaluación. Puesto que el prototipo se iba a ejecutar en máquinas multicore parece lógico intentar aprovechar de alguna manera el paralelismo que ello nos ofrece. Para ello se eligió un lenguaje basado en el paradigma PGAS (ver 2.1.4) que ofrece de forma natural ventajas cuando se trabaja con datos locales al nodo.

Puesto que cada nodo puede contener infinidad de documentos y todos estos resultados han de viajar al nodo encargado del ranking definitivo de la query, es necesario limitar el tamaño del conjunto de resultados. Por ello definiremos un valor K que determina

el tamaño máximo del conjunto de resultados [15] (y el tamaño que podría ser enviado por cada nodo).

La tarea se puede optimizar más si consideramos que los documentos están uniformemente distribuidos entre los P procesos y por tanto se puede optar por presentar un esquema iterativo en el que en cada iteración los P procesos envían cada uno $\frac{K}{P}$ documentos al nodo ranker para formar un total de K resultados que pueden formar una respuesta a la query. Si se considera que los resultados enviados no satisfacen la query se puede pedir otra iteración al resto de los nodos. Aunque en el caso peor podrían necesitarse P iteraciones en el caso medio rara vez se necesitan más de dos iteraciones. Los resultados experimentales indican que el valor más adecuado de K para casi todos los casos es 128.

Capítulo 2

Modelo de programación PGAS

2.1. Modelos de programación paralela

En esta sección se describien los modelos de programación paralela más comunes.

2.1.1. Modelo con paralelismo de datos

En este modelo se persigue que cada unidad de proceso en el sistema ejecute las mismas tareas sobre una parte del subconjuto total de datos. Puede haber un solo hilo de ejecución que controle las operaciones en todos los fragmentos de datos o varios hilos controlando su parte de la operación pero todos ejecutando el mismo código.

Puesto que en este modelo todos los procesos ejecutan el mismo código sobre un fragmento de datos independiente del resto, es un modelo sencillo de programar y que no requiere apenas mecanismos de sincronización. Su principal desventaja reside precisamente en que el codigo es el mismo en todas las tareas y por tanto todos los nodos de proceso deben realizar la misma operación.
Un lenguaje de programación muy conocido basado en este modelo es HPF *High Performance Fortran*. Fue publicado por primera vez en 1993 por el *High Performance Fortran Forum*. Permitia implementaciones SIMD y MIMD muy eficientes pero debido a la dificultad en distintos aspectos de la implementacion fue abandonado por la mayoria de vendedores. A pesar de todo sigue teniendo mucha influencia en lenguajes actuales.

2.1.2. Modelo de paso de mensajes

En este modelo la comunicación se basa en el concepto de mensaje. Un mensaje es la versión de alto nivel del concepto de datagrama y representa una acción tal como la invocación de una función, señales o intercambio de datos. Es un modelo que ha recibido

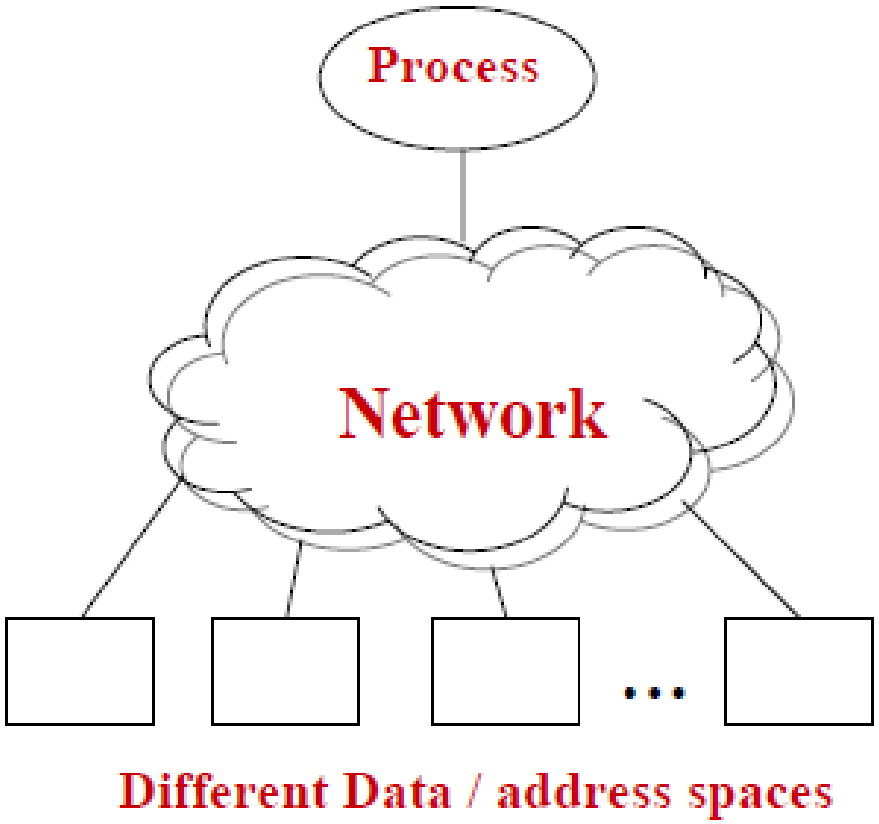

Figura 2.1: Modelo de paralelismo de datos

el nombre de *"shared nothing"* ya que la abstracción de paso de mensajes oculta cambios de estado internos a la implementación.

Las principales ventajas de trabajar con este modelo de computación parelela son tanto el control total, por parte del programador, del flujo de datos y de la distribución del trabajo, como la abstracción completa de los modelos internos. El precio a pagar puede ser un *buffering* excesivo, así como una sobrecarga en el número de mensajes enviados cuando se realizan múltiples operaciones pequeñas.

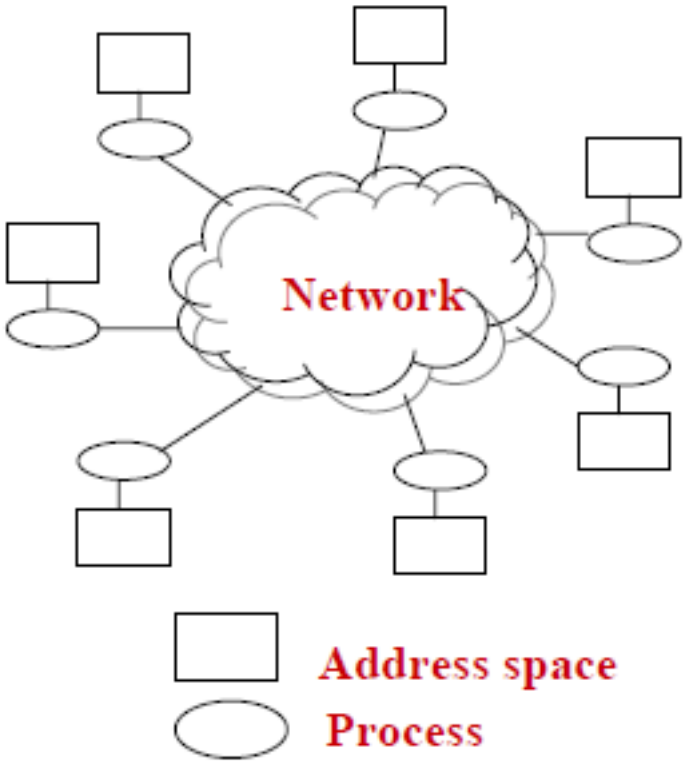

Figura 2.2: Modelo de paso de mensajes

El lider actual en sistemas de paso de mensajes es *Message Passing Interface* (MPI). MPI es un protocolo de computación paralela independiente del lenguaje. Suele estar disponible como bibliotecas que son llamables desde lenguajes como C o Fortran. Actualmente hay dos modelos en uso de MPI

- MPI-1 : No contiene ningun concepto de memoria compartida. El forzar todo el trabajo con paso de mensajes incita al programador a aprovechar lo mas posible la localidad en memoria y por tanto tiende a hacer implementaciones NUMA más eficientes.

- MPI-2 : Incorpora algunos conceptos de memoria compartida así como diversas funcionalidades *One-Sided* y una implementación de Entrada/Salida paralela

2.1.3. Modelo de memoria compartida

En el modelo de memoria compartida el acceso a los datos se realiza de manera simplificada mediante expresiones de lectura/escritura que son similares (o identicas) a las expresiones locales.

La ventaja mas evidente es que para el programador los accesos locales o remotos suelen ser indistiguibles y, en ese aspecto, no necesita diferenciarlos para tener una tarea que funcione correctamente. Sin embargo no está exento de desventajas como son: la necesidad de usar mecanismos de sincronización a la hora de manipular datos compartidos, la imposibilidad de aprovechar la localidad espacial y, en general, que es más facil diseñar programas poco eficientes por la falta de diferencia entre los accesos locales y globales.

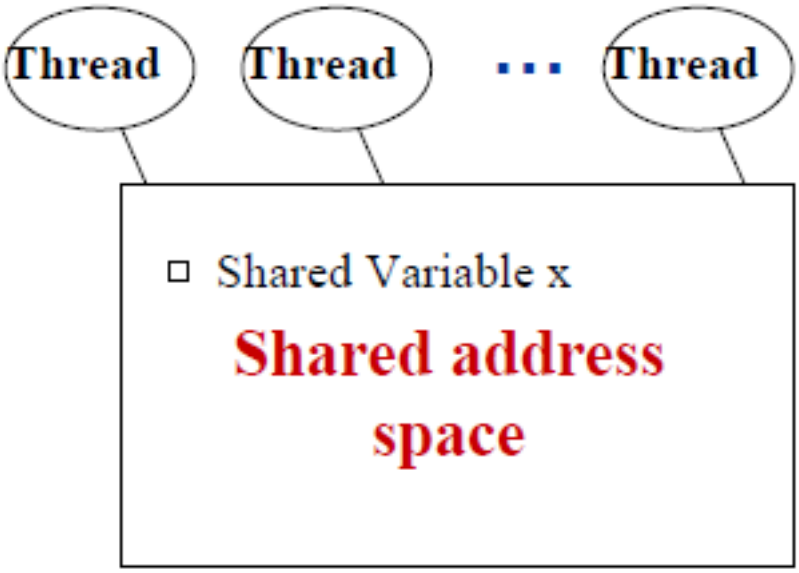

Figura 2.3: Modelo de paso de mensajes

Una de las implementaciones mas usadas del modelo de memoria compartida es *Open Multi-Processing* (OpenMP). OpenMP ha sido definido en conjunto por un grupo de fabricantes de software y hardware. Es un API multiplataforma, portable y escalable para programación paralela con memoria compartida que, mediante directivas de compilación, rutinas de biblioteca y variables de entorno, permite influir en el modo de ejecución. Suele combinarse con MPI o con extensiones OpenMP para poder actuar en sistemas híbridos que usan tanto memoria compartida como distribuida.

2.1.4. El modelo PGAS

Las siglas PGAS se corresponden con el modelo de programación paralela **Partitioned Global Address Space** (Espacio de Direcciones Global Particionado) también conocido como *modelo de memoria compartida distribuida*. En este modelo se asume la existencia de un espacio de direcciones de memoria global que está particionado de forma lógica (y en ciertos aspectos también de forma física ya que distintos bloques de memoria pertenecen físicamente a distintos nodos de proceso) y en el cual cada porción de éste es local a un procesador.

Una de las novedades respecto a otros modelos de programación paralela con espacios de memoria compartida es que las distintas porciones del espacio de direcciones pueden tener *afinidad* con un hilo de ejecución en particular, lo cual permite aprovechar la localidad de las referencias.

Puesto que la motivación del proyecto es diseñar un motor de búsqueda usando un lenguaje basado en este paradigma (UPC) dedicamos la siguiente sección a hablar más en profundidad de PGAS y de los lenguajes que hacen uso de este modelo.

2.2. El modelo de espacios de direcciones global particionado

El modelo PGAS está inspirado en los sistemas de memoria compartida tal y como se ha comentado en 2.1.4. Su idea principal es distribuir el espacio de direcciones en fragmentos que poseen afinidad por un thread determinado. Por tanto permiten aprovechar la localidad espacial de los datos resolviendo referencias locales de forma optimizada. Al igual que en el modelo de memoria compartida, las expresiones de acceso son sencillas y similares a las usadas en lenguajes monoproceso.

2.2.1. Breve reseña histórica

Históricamente existian muchos lenguajes de computación paralela. En un momento determinado empiezan a surgir nuevas alternativas basadas en en el concepto de memoria global/compartida (Split-C,F–,CC++...) que finalmente llevaron a la definición del modelo PGAS que tenemos actualmente. También han tenido especial influencia en los modelos PGAS el modelo BSP (que se basa en la alternancia entre computación e intercambio de datos) y el *Global Arrays* que bajo la apariencia de un toolkit ya incluia conceptos de localidad.

2.2.2. La visión de un programa PGAS

Un programa escrito en un lenguaje que soporte el modelo PGAS debe ajustarse a la siguiente visión:

- Es un solo programa que se ejecuta en cada nodo de computación.
- El programa tiene multiples hilos de control (normalmente 1 por nodo de computación físico o por thread).
- Presenta un bajo grado de virtualización

2.2.3. La visión de memoria para un programa PGAS

El modelo PGAS separa la memoria en dos zonas bien diferenciadas:

- La zona "compartida" que puede ser referenciada por cualquier thread de la aplicación.
- La zona privada a la cual solo tiene acceso el thread local donde ha sido reservada.

La existencia de dos zonas "distintas" de memoria implica la existencia de distintos tipos de punteros. Los punteros pueden residir en memoria compartida o privada y apuntar a memoria compartida o privada.

2.2.4. Sincronización básica para un programa PGAS

Al igual que un programa de memoria compartida , el modelo PGAS precisa en ocasiones del uso de métodos de sincronización en el acceso a datos. Los métodos de sincronización más usados en este ámbito son:

- Barreras: provocan una interrupción de la ejecución hasta todos los procesos llegan a este punto del códg. Algunos lenguajes soportan barreras de fase dividida que permiten separar el punto de parada del punto de continuación.
- Funciones colectivas que realizan una operación sobre un conjunto de datos compartidos.
- Mecanismos de ordenación de escrituras: a veces es necesario asegurar la consistencia secuencial en el acceso a datos y en otras no nos importa tanto. Los lenguajes PGAS suelen implementar un modelo de consistencia relajada en el que no se garantiza el la consistencia secuencial. Por ello suelen estar disponibles para

el programador primitivas comoo las *b*arreras de memoria (fences) que fuerzan en puntos determinados del código la consistencia secuencial.

- Cerrojos y Semáforos: su uso al igual que en la programación concurrente clásica permite sincronizar el acceso a datos y forzar el orden.

2.2.5. Rendimiento para un programa PGAS

En un programa que use este paradigma suele ser un factor muy importante a la hora de hablar de rendimiento la correspondencia entre el modelo y la arquitectura física de las máquinas que se utilizan. En caso de que la correspondencia sea mala las penalizaciones de rendimiento son muy elevadas (mensajes pequeños en una Ethernet o intentar aprovechar un acceso a datos de gran ancho de banda mediante mecanismos de paso de mensajes).

Así mismo es fundamental diseñar un modelo de aplicación que se ajuste lo más posible al modelo PGAS. Ésto en ocasiones puede llegar a ser muy complicado y es uno de los problemas clave en la creación de programas multiproceso usando este paradigma.

2.2.6. Lenguajes basados en el modelo PGAS

A continuación presentamos algunos de los lenguajes basados en el paradigma de memoria compartida distribuida más conocidos:

Co-array Fortran

Previamente conocido como F– es una extensión para Fortran 95/2003 con soporte para procesamiento parelelo. Los programas Co-array Fortran se interpretan como si hubieran sido replicados un cierto número de veces y todas las copias se hubieran ejecutado de forma asíncrona. Cada copia posee su propio conjunto de datos de trabajo y se la denomina imagen. El comité ISO Fortran decidió en 2005 incorporar *co-arrays* en la versión 2008 de Fortran Estandard.

Para más información se puede recurrir a [2]

Fortress

Fortress es una especificación para un lenguaje paralelo basado en Fortran que fue inicialmente desarrollado por Sun Microsystems. Su nombre viene de *Secure Fortran* aunque sintáticamente no tiene demasiado parecido con éste sino que se parece mas a

ML o Haskell. Tiene soporte integrado en sus bibliotecas (basadas en Java) de computación paralela y permite redefinir estructuras del lenguaje (como el bucle loop) de la forma que decida el usuario. Tiene soporte UNICODE y su sintaxis intenta asemejarse en cierta medida al lenguaje matemático.

Durante un tiempo de dudó de que el proyecto pasara de ser un borrador pero en 2008 salió a la luz la versión 1.0 de la especificación junto con su implementación correspondiente. Ésta se puede encontrar en [3]

Chapel

Chapel [4] es un lenguaje para procesamiento paralelo diseñado por Cray como participación en el programa HPCS (High Producitivty Computing Systems, financiado por DARPA) destinado a incrementar la productividad en supercomputación. Los objetivos de este lenguaje que lo desmarcan de los demás son: facilitar al usuario un nivel de expresión más elevado y tratar de separar las expresiones algorítimicas de los detalles de implementación de datos. Soporta además conceptos de orientación a objetos y de tipos genéricos.

X10

X10 es otro lenguaje de programación que está siendo desarrollado dentro del programa HPCS de DARPA por IBM.

Es un lenguaje que permite usar tanto programación orientada a objetos como programación procedimiental clásica. Su base es un subconjunto de las funcionalidades de Java con ciertos añadidos para el trabajo con arrays y la concurrencia. Además incorpora el concepto de "relaciones padre-hijo" entre tareas para evitar las situaciones de interbloqueo que se producen cuando dos o más procesos esperan entre si a que finalicen. Una tarea hija por tanto no puede esperar nunca a que acabe una padre.

Para más información se puede acudir a la página web del proyecto X10 [5]

Unified Parallel C

Unified Parallel C (UPC) es una extensión paralela explícita de ANSI C y está basado en el modelo de programación de espacios de memoria globales y particionados, también conocido como el modelo de programación de memoria compartida distribuida.

UPC mantiene los potentes conceptos y características de C y añade paralelismo, acceso a memoria global con una compresión de lo que es remoto y de lo que es local, y la habilidad de leer y escribir en memoria remota con simples declaraciones. UPC se

basa en la experiencia obtenida compiladores de leguaje C con memoria compartida distribuida anteriores como son Split-C, AC, y PCP. La simplicidad, la facilidad de uso y rendimiento de UPC han despertado el interés de los usuarios y proveedores de la computación de alto rendimiento. Este interés ha está dando lugar al desarrollo de proveedores y la comercialización de compiladores de UPC.

UPC es el esfuerzo de un consorcio del gobierno, la industria y el mundo académico. A través del trabajo y las interacciones de esta comunidad, la especificación V1.0 de UPC fue producida en Febrero de 2001. Posteriormente, la especificación V1.1 de UPC fue lanzada en mayo de 2003. Hay implementaciones comerciales para plataformas de HP, SGI, Sun y Cray aunque también se pueden encontrar implementaciones de código abierto disponibles en la Universidad Tecnológica de Michigan (MuPC) para las arquitecturas de 32bit de Intel y la Universidad de California de Berkeley (BUPC) para varias plataformas incluyendo las arquitecturas de 32bit y 64bit de Intel. Asimismo existe una implementación de código abierto para el Cray T3E así como muchas otras implementaciones en marcha.

La información recogida en esta sección sobre UPC puede ser ampliada en [6] y [7]

El modelo de memoria de un programa UPC es el clásico del modelo PGAS (cada proceso tiene su zona de memoria local y acceso a memoria compartida que está distribuida entre todos los nodos). Cada fragmento de memoria compartida posee afinidad con un cierto thread, lo cual implica que reside en memoria local para ese thread y por tanto se pueden optimizar los accesos al fragmento.

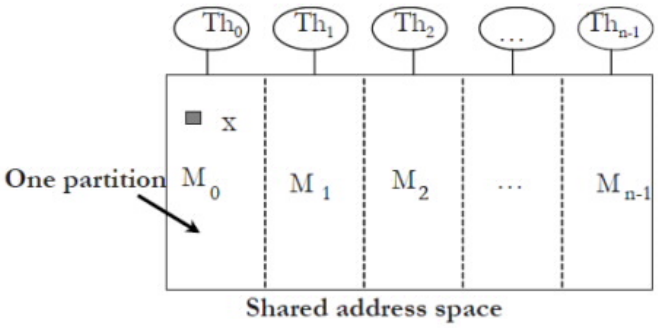

Figura 2.4: Modelo de memoria compartida distribuida

La distribución de datos en memoria se realiza mediante una política round-robin de tamaño definido por el programador. El acceso a zonas de memoria compartida remotas a través de un puntero remoto tiene la misma sintaxis que el acceso por puntero (o array) de C. Sin embargo hay que tener en cuenta que la memoria está efectivamente particionada y no existen operaciones que trabajen con zonas de memoria de diferente afinidad (es decir, no existe un "strcpy" que trabaje con datos de varias afinidades

simultaneamente)

Los punteros compartidos son el método de acceso standard a los datos situados en memoria compartida. Un puntero compartido consta de 3 campos diferenciados :

- Un campo dirección: que contiene una dirección virtual.
- Un número de bloque o *fase*: indica a cual de los bloques repartidos según la política round-robin apunta el puntero.
- El indicador de afinidad del puntero: contiene un entero que representa la afinidad del dato apuntado por el objeto.

Los punteros compartidos no contienen información sobre el tamaño del bloque, por lo tanto, cualquier acceso por puntero y toda la aritmética de punteros compartidos ha de ser resuelta en tiempo de compilación (a pesar de que el tamaño de reparto de bloque se puede definir de forma dinámica), lo cual en ocasiones resulta ciertamente complejo.

La implementación de los punteros compartidos varía segun el fabricante del compilador. En el caso de la implementación de la universidad de Berkeley (la usada en este proyecto) se soportan 3 tipos de punteros compartidos:

- Empaquetados: ocupan un entero de 64 bit. Son muy eficientes pero los bloques de memoria compartida tienen tamaño reducido.
- En estructura: están representados en un `struct` clásico de C. A cambio de poder referenciar bloques más grandes tienen una penalización de rendimiento en ejecución, por lo que suelen evitarse en entornos de producción.
- Simétricos: solo están disponibles en determinadas arquitecturas de memoria compartida y presentan un rendimiento semejante a los nativos de C.

Respecto a los métodos de sincronización disponibles en UPC hay que reseñar que en su versión 1.2 son bastante limitados. Se ofrecen al programador *fences*, barreras normales y de fase dividida así como cerrojos. Aunque con estas primitivas de sincronización se pueden realizar prácticamente la mayoría de tareas, otras muchas se vuelven incómodas de programar o tienen pérdidas de rendimiento importantes (los cerrojos tienen penalizaciones importantes). Por ello los implementadores de compiladores de UPC han ido añadiendo extensiones que se alejan del standard. En concreto las extensiones BUPC usadas en el proyecto facilitan semáforos y métodos de transferencia asíncrona que facilitan no solo la sincronización sino que, en parte, solventan el problema del acceso a regiones virtualmente contiguas pero situadas en distintas afinidades. [10] [9].

El lenguaje además se complementa con bibliotecas que contienen funciones colectivas y otras para E/S paralela que simplifican determinadas tareas al programador.
Para probar el comportamiento de UPC hemos desarrolado una versión del algoritmo *NMF (Non negative Matrix Factorization).*
Este es un algoritmo que, por sus propiedades, se presta como caso ideal para las pruebas del lenguaje UPC. El funcionamiento simétrico de su algoritmo, así como las propiedades de las operaciones con matrices, hacen que la división de la memoria en cada uno de los procesadores sea adecuada para distribuir las matrices a través de los procesadores.

Para empezar, explicaremos por encima la utilidad del algoritmo y enunciaré la distribución de las variables en la memoria, y posteriormente, comentaré la optimización necesaria para no cargar excesivamente la comunicación entre procesadores.

El objetivo del algoritmo consiste en, dada una matriz de tamaño n x m, hallar otras dos matrices W y H de tamaño n x k y k x m respectivamente que, multiplicadas entre sí, devuelvan la matriz original. Consiste en la repetición iterativa de una secuencia de cálculos que tras una cantidad de pasos (o hasta que se satisfaga un criterio de convergencia entre la matriz original y la obtenida por la multplicación de W y H) obtendrá las matrices que factorizan la original.

Las matrices W y H sobre las que se trabajará se inicializan aleatoriamente con valores positivos. La matriz V es la matriz original que se quiere factorizar.

El algoritmo se divide en dos cálculos: el de la matriz W y la matriz H. Sus pasos son idénticos, salvo por la división de la memoria.

```
//Matriz V
shared [n*m/THREADS] double Vw [n*m];
shared [m/THREADS] double Vh [n*m];
```

Es la matriz original. Se copia en dos variables porque cada una de ellas está distribuida de distinta forma. Para ilustrar los ejemplos utilizaremos tamaños de matrices concretos: n, m =4; k=2; Threads=2.

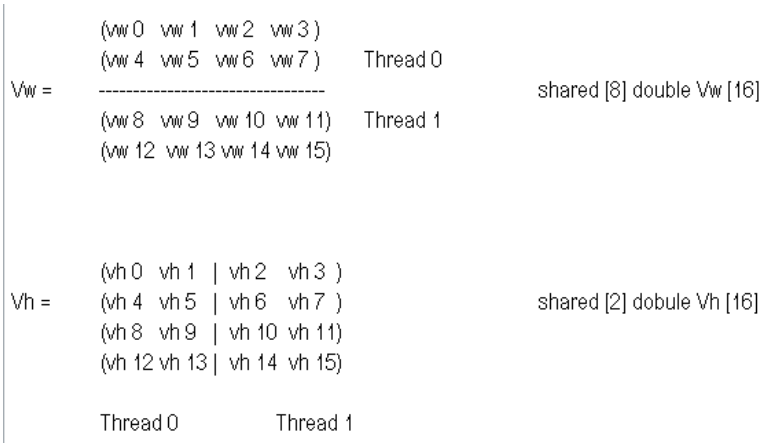

Figura 2.5: Matrices V en NMF

double WHa [n*m/THREADS]; Es una matriz auxiliar para hacer un cálculo. Equivaldría a tener una variable shared de la siguiente forma: shared [n*m/THREADS] double WHa [n*m], pero como ningún procesador necesitará consultar una parte de la matriz que no haya calculado él mismo, podemos utilizar una matriz no compartida sensiblemente más pequeña. En general, se han declarado tantas variables no compartidas como ha sido posible. double Wa [n*k/THREADS];

```
//Matriz W
shared [n*k/THREADS] double W [n*k];
shared [n/THREADS] double Wt [k*n];
shared [k/THREADS] double AccH [1 * k];
```

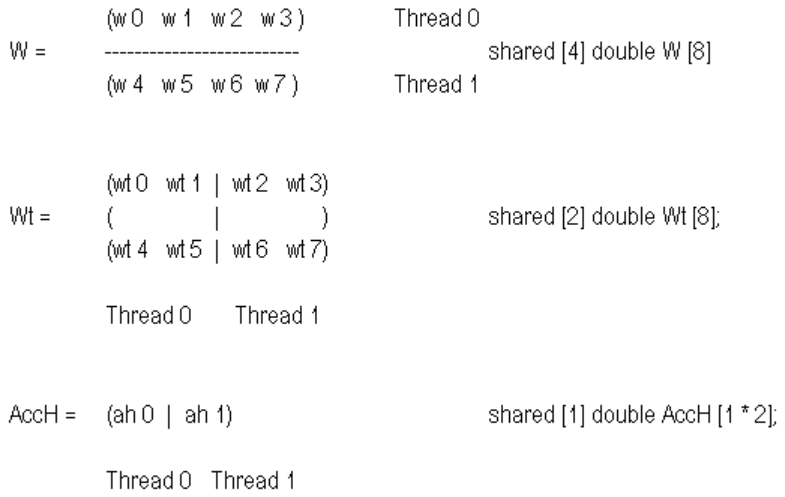

Figura 2.6: Matrices W en NMF

```
//Matriz H
double Ha [k*m/THREADS];
shared [m/THREADS] double H [k*m];
shared [m*k/THREADS] double Ht [m*k];
shared [k/THREADS] double AccW [k * 1];
```

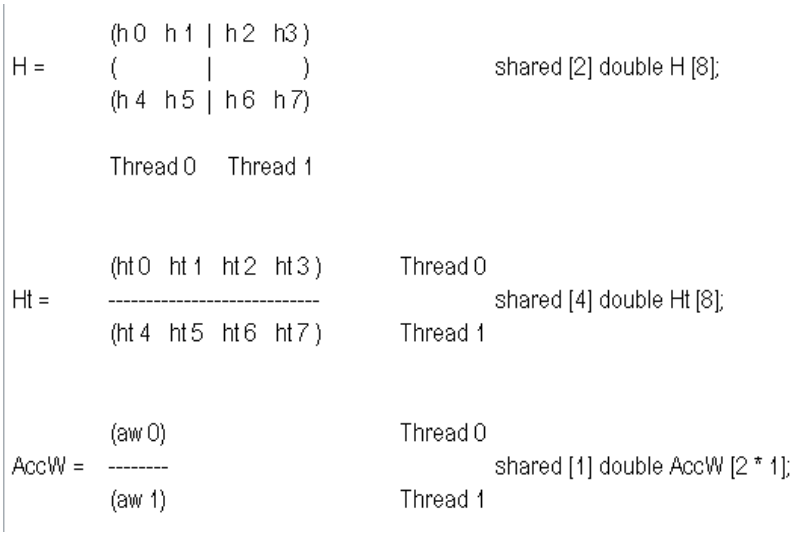

Figura 2.7: Matrices H en NMF

El algoritmo original ha tenido que sufrir optimizaciones a causa de un problema. Este problema ha sido recurrente cada vez que había que multiplicar dos matrices:
Matriz1 = Matriz2 * Matriz3
Pongamos, para simplificar el ejemplo, que trabajamos con matrices cuadradas y que las matrices 1 y 2 están distribuidas de la misma forma en memoria:

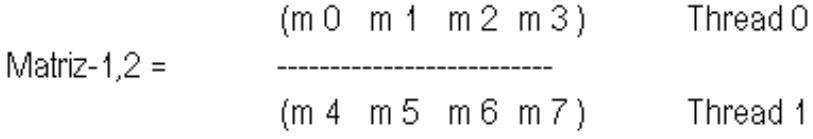

Figura 2.8: Matrices 1 y 2 en NMF

Y la matriz 3 distribuida por columnas:

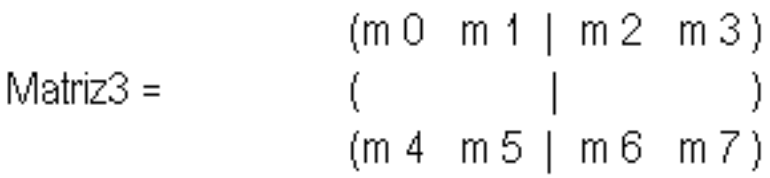

Figura 2.9: Matriz 3 en NMF

En este caso hay un problema: para hacer la multiplicación cada parte local tiene que consultar la Matriz3 entera. Como intentar acceder a cada elemento de la matriz según se va necesitando cargaría excesivamente el tráfico entre los procesadores (salvo para aquellos elementos que ya están en el procesador local), la solución empleada en este caso fue traerse de golpe cada trozo de la matriz que reside en cada procesador. Así, el número de accesos que ha de hacer cada procesador a memorias ajenas es el número de procesadores involucrados en el cálculo (excepto el trozo que reside en memoria local).

Los ejecución del algoritmo escala según el número de procesadores. Según podemos observar en la siguiente gráfica, con valores de k fijos, y duplicando tanto el tamaño de la matriz a calcular como el número de procesadores, el tiempo invertido en el cálculo es casi constante. Los resultados se muestran en la siguiente figura (en el eje Y el tiempo en segundos de ejecución)

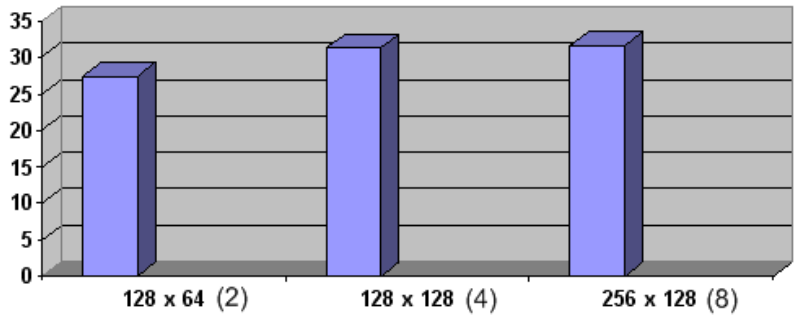

Figura 2.10: Resultados de NMF para k=8 duplicando el tamaño de problema

En cambio, si cuadruplicamos el tamaño de la matriz a calcular, los tiempos obtenidos escasamente se duplican:

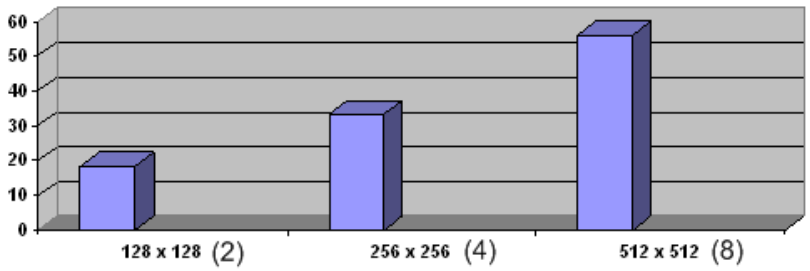

Figura 2.11: Resultados de NMF para k=16 cuadruplicando el tamaño de problema

Parte III

Implementación y resultados

Capítulo 3

Implementación de un buscador para operaciones de tipo OR en escenarios de tráfico elevado

3.1. Operaciones de lectura con escrituras offline

3.1.1. Introducción

Como parte fundamental del proyecto se decidió realizar un motor de búsqueda de tipo OR sobre UPC. En concreto nos hemos centrado en la parte del ranker y presuponemos que unos hipotéticos modulos indexadores y broker han generado los índices de términos/documentos y las consultas que se van a ejecutar. Como decisión de diseño se ha decidido optimizar el programa para sostener un *throughput* de consultas alto en situaciones de tráfico elevado. Así mismo intentaremos obtener el volumen de tráfico óptimo para el buscador.
Respecto a las escrituras en esta versión consideraremos que no tiene importancia su ejecución de forma inmediata y por tanto se realizarán en algun momento de forma offline de forma desacoplada con nuestro buscador.

3.1.2. Pruebas preliminares

La versión completa del buscador de solo lectura se corresponde con la v7 del código entregado junto con el proyecto. Las 6 versiones anteriores se corresponden a tomas de contacto con UPC, primeras pruebas de rendimiento, evaluación de la viabilidad de usar las extensiones BUPC y su impacto en el funcionamiento del programa, así como comprobar la efectividad de distintos modelos de datos en la ejecución del motor.

3.1.3. Distribución de los datos

Una de las decisiones de diseño más importantes ha sido elegir un modelo de datos que permita conseguir un alto rendimiento en lecturas ya que estas ocupan prácticamente el 100 % del tiempo de proceso. Además como se comentó en 2.1.4 en el modelo PGAS es fundamental elegir correctamente un modelo de datos que se corresponda con el dominio del problema y además sea adecuado para la arquitectura usada.

Posting list

Para las posting list se ha elegido la representación más simple y eficiente en lo que se refiere a lecturas consecutivas de datos: un array . Este array es cargado en la fase de inicialización con todos los documentos contenidos en la base de datos y proveerá de la información necesaria para que el algortimo de Ranking pueda producir los resultados adecuados.

Aunque lo ideal sería repartir la memoria compartida reservada para los D documentos del índice en P bloques de tamaño $\frac{D}{P}$, esto no es posible pasado un cierto número de documentos $K * 2^{16}$ [1]ya que el formato de puntero empaquetado presenta esa limitación en tamaño de bloque. Por tanto las divisiones se hacen en bloques de 32K documentos y se han generado macros auxiliares para traduccir de direcciones lineales virtuales dentro de un bloque de tamaño $\frac{D}{P}$ a direcciones reales segun la división real (ver fig. 3.1).

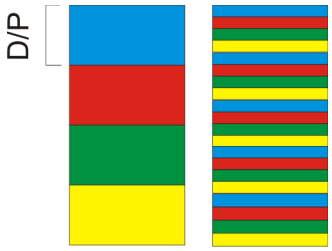

Figura 3.1: Modelo ideal de distribución de las posting list vs. distribución adaptada a las limitaciones de los punteros empaqueados

En condiciones normales esto supone una penalización sólo en determinados términos que necesitan en alguna iteración del algoritmo de ranking una llamada de copia de memoria extra.

Para permitir una buena escalabilidad, los documentos de cada término se reparten según una política *round-robin* de tamaño $\frac{K}{P}$ (Figura 3.2) que encaja de manera simple con el modelo de datos que espera recibir el algoritmo de ranking. Para cada término se empieza el reparto por un nodo distinto de manera que la información esté distribuida

[1]Este tamaño es consecuencia de una limitación propia de la implementación de UPC utilizada

de la forma más uniforme posible (si no, por la ley de Zipf [11] [12], la mayoria de documentos serían asignados a los primeros Threads). Además se garantiza que para cada thread la información para un cierto término se almacena en el array de forma secuencial (exceptuando las limitaciones que el uso de punteros empaqueados nos impone en el tamaño de las divisiones de memoria) lo cual reduce el número de peticiones de datos al poderlas agrupar.

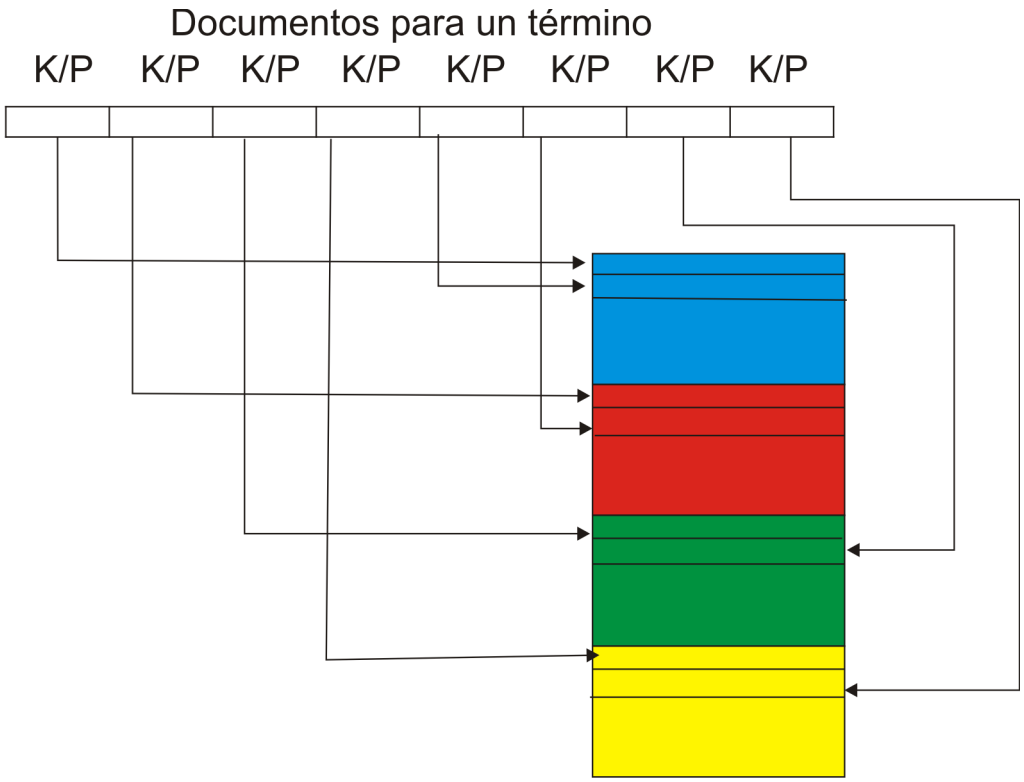

Figura 3.2: Vision en modelo ideal del reparto de documentos por término

Cada elemento del array es una estructura de tipo Documento declarada de la siguiente forma:

```
struct Documento
{
        int idDocumento;
        float frec;
}
```

Y que define el identificador del documento y su frecuencia de aparición. Con estos dos valores se puede generar respuestas a las consultas de forma eficiente mediante el algoritmo que explicaremos más adelante en la seccion 3.1.6.

Lista de términos

La lista de términos contiene cada uno de los términos para los que nuestro buscador tiene documentos asociados. A diferencia de las posting list se ha decidido que cada thread tenga una copia local de la lista de términos. Los motivos han sido los siguientes:

1. Las listas de términos tienen un tamaño considerablemente menor que las posting list. Por eso no se consigue un aumento de la escalabilidad apreciable repartiendolas entre los nodos de proceso
2. Estas listas son accedidas múltiples veces por el nodo ranker de una consulta dada para cada iteración de cada término de la consulta. Esto podría suponer una penalización de rendimiento importante en caso de que la lista residiera en memoria remota.

A nivel de implementación la lista es un simple array de una estructura *Termino* que pasamos a concretar y a detallar

```
struct Termino
{
        int idTermino;
        int inicio[THREADS];
        int fin[THREADS];
}
```

El campo idTermino se corresponde con el identificador único del término y es el valor que aparece en una query. Los dos arrays *inicio y fin* contienen los índices lineales (hay que aplicarles las macros correspondientes para transformarlos en indices reales) de las posiciones del array compartido de documentos donde empiezan y terminan respectivamente los documentos relativos al término en los bloques de memoria compartida asignados a cada thread.

Aprovechando que los términos están ordenados y contiguos en los ficheros de índice y que las escrituras se realizan de forma offline se ha podido hacer coincidir el idTermino con su posición en la lista de términos y por tanto los accesos a un término dado se realizan en O(1)

Consultas

Puesto que el desarrollo del motor se ha centrado en el módulo de ranking se ha decidido simular la llegada de consultas desde el broker mediante una cola de consultas que es leida desde un archivo en la fase de inicialización. La estructura usada para almacenar una consulta desde su lectura desde disco hasta que queda completamente resuelta es la siguiente:

```
struct Query
{
        int idQuery;
        int numTerminos;
        int terminos[MAX_TERMINOS_QUERY]
        int vecesTerminos[MAX_TERMINOS_QUERY]
        Documento* resultadosQuery[MAX_TERMINOS_QUERY];
        int numResultados[MAX_TERMINOS_QUERY];
        mw_mapa* iteraciones;
        mw_mapa* freq_terms;
        int* lista_docs;
        float* lista_frecs;
        int totalResultados;
        int salidaForzada;
}
```

Donde el significado de cada campo es el siguiente:

- idQuery : Contiene el identificador único asociado a una query
- numTerminos : Cantidad de términos a buscar que contiene la query
- terminos: Identificadores de los términos a buscar
- vecesTerminos: Número de veces que aparece un término dado en la query
- resultadosQuery : Buffers para almacenar los documentos necesarios para cada iteracion
- iteraciones: Wrapper para un hashmap de STL que es usado por la clase de ranking y que debe preservarse entre iteraciones.
- freq_terms: Wrapper para un hashmap de STL que es usado por la clase de ranking y que debe preservarse entre iteraciones.
- lista_docs: Buffer que es reservado por el ranker y donde se almacenan los identificadores de documento que forman parte de la respuesta a una query.
- lista_frecs: Buffer que es reservado por el ranker y donde se almacenan las frecuencias de documento que forman parte de la respuesta a una query.
- totalResultados: Número de resultados devueltos por el módulo de ranking.
- salidaForzada: Booleano que indica si se ha forzado una salida del módulo de ranking para esta consulta. La salida se fuerza si el algoritmo determina que necesita más datos pero sin embargo no hay más documentos pendientes de ser analizados.

Otros datos usados por el ranker

Aparte de los datos mencionados en las secciones anteriores, el módulo de ranker precisa de otros datos para realizar su función de forma corriente. Estos datos debido a su uso intensivo se encuentran replicados por cada thread y almacenados en las estructuras requeridas por el ranker durante la fase de inicialización. Se comentará más sobre estos datos en la sección dedicada al algoritmo de ranking pero una visión general de los mismos sería:

- Una lista de palabras junto con el idTermino que le ha sido asociado
- Una lista de pesos para cada uno de los términos existentes
- Un conjunto de palabras consideradas *stopwords*

3.1.4. Comunicación entre threads

Aunque con el modelo actual de datos no es necesaria comunicación alguna (aparte de las transferencias de datos gestionadas por UPC) para realizar la tarea de Ranking correctamente se decidió implementar un sistema de paso de mensajes simples con vistas a futuras versiones que probablemente necesitarían algun tipo de comunicación entre hilos de ejecución. En esta versión el sistema de mensajes se ha usado exclusivamente para que un thread comunique a los demás que ha terminado de procesar todas las consultas que tenía asignadas.

Para su implementación se ha usado una cola local de mensajes para desacoplar la acción de encolar el mensaje de la de intentar mandarlo propiamente dicha evitando así bloqueos cuando el destinatario tiene su buffer de mensajes lleno. Los buffer de mensaje se han implementado mediante un array compartido de P elementos de tipo BufferMensaje distribuidos con un round-robin de 1 (cada thread tiene el suyo).

```
struct BufferMensajes
{
        Mensaje mensajes[TAM_BUFFER_MENSAJES];
        int nMensajes;
        int posE;
        int posL;
        upc_lock_t* lock;
}
```

Para la sincronización de acceso a cada buffer se ha recurrido al uso de un cerrojo de upc (campo *lock* de la estructura BufferMensajes). Se ha intentando minimizar el número de operaciones con cerrojos ya que,como se comenta en 2.2.6 y [10], los cerrojos de UPC tienen una penalización de uso muy importante.

3.1.5. Funcionamiento del programa

La visión general de funcionamiento del programa se ajusta a la figura 3.3 y será detallada a continuación.

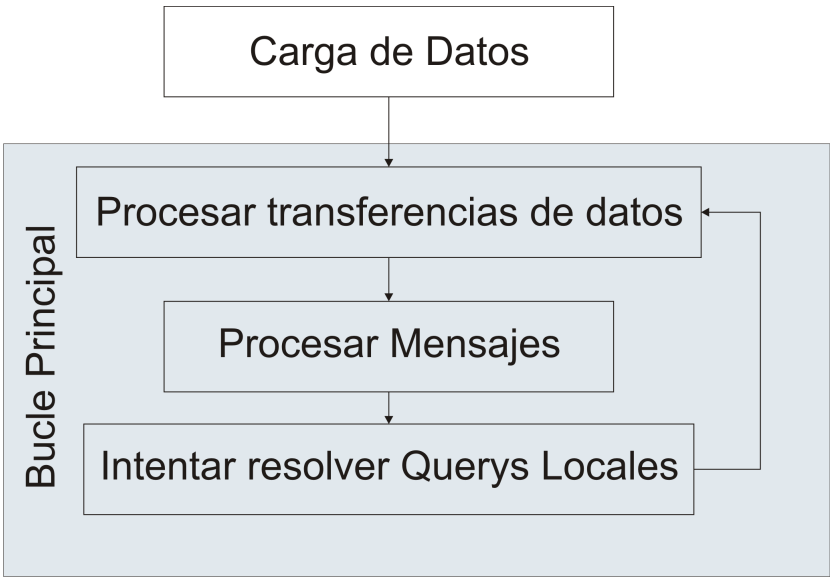

Figura 3.3: Flujo del programa

Carga de datos

A pesar de que el proceso de carga e inicialización de datos no tiene influencia en la medida de rendimiento del buscador, en nuestro caso era fundamental conseguir una inicialización lo suficientemente rápida como para poder realizar pruebas con parámetros diversos y poder intentar depurar los errores que han ido surgiendo.

En un principio se pensó que nos podrían ser útiles las rutinas de I/O paralela que nos ofrece UPC. Sin embargo el modelo de datos usado y el formato de los archivos de índice no permitieron su uso. En las primeras versiones de prueba optamos por una modelo de inicialización en el que el Thread 0 cargaba todos los datos y los distribuia entre el resto de threads. Desafortunadamente el proceso duraba demasiado y no nos podíamos permitir perder ese tiempo cada vez que necesitábamos realizar una prueba. Finalmente optamos porque todos los threads realizaran la inicialización simultaneamente partiendo de los mismos ficheros de datos pero obviando los datos, que no debían almacenar. De esa manera logramos un incremento de la velocidad de carga de aproximadamente un 500 %

Gestión de las transferencias de datos

Puesto que las transferencias de datos son gestionadas por UPC nuestra tarea consistia en agrupar las peticiones de transferencia buscando minimizar el tiempo en que el sistema de ranking estaba *idle* porque no disponía de datos.

El proceso de transferencia de datos se compone de los siguientes pasos:

1. Si hay una transferencia de datos en curso revisar si ya ha concluido. Si es así asignar las querys que han recibidos datos nuevos a la cola de preparadas para su posterior procesado.
2. Si no la hay, pero tenemos suficientes consultas en la cola de "pendientes de recibir datos" se procede a realizar una petición de datos al resto de threads. Si el numero de querys en este caso no llega al tamaño de lote configurado se añadirán querys de la cola inicial hasta llegar al tamaño deseado.

El proceso de petición de datos se aprovecha de las extensiones BUPC para transferencia multi-origen definidas en [9]. En un primer paso se analizan todas las querys que estan preparadas para recibir datos para determinar las direcciones de memoria de donde van a recibir los datos. Una vez se han establecidos todos los orígenes de datos una sola llamada a *bupc_vlist_memget_async* inicia una petición asincrona de copia de datos de memoria compartida a local que obtendrá todos los datos necesarios para realizar una iteración para cada una de las querys del lote. Al ser una llamada asíncrona el programa puede seguir procesando querys para las que sí tiene datos y responder a mensajes de otros threads.

Procesado de mensajes

En esta fase el programa intenta procesar un mensaje entrante si existe y posteriormente mandar todos los mensajes que tenga en la cola de salida. Este proceso como se ha comentado en su sección es no bloqueante.

Procesado de las querys

Si al llegar a esta fase del bucle principal tenemos alguna query con datos suficientes para ser procesada, llamamos al algoritmo de ranking para realizar una iteración sobre ella. Una vez completada la iteración el algoritmo nos puede comunicar tres estados distintos:

1. El algoritmo ha determinado que posee documentos que se ajustan de forma suficiente a la query. La query por tanto está resuelta y podría ser enviada de vuelta

al broker para que se la facilite al usuario. En nuestro caso ante la inexistencia de un broker nos limitamos a escribir los resultados en un fichero de log para depuración.

2. El algoritmo ha determinado que necesita más datos. La query vuelve a pasar a la cola de pendientes para realizar otra iteración sobre ella.

3. El algoritmo ha determinado que necesita más datos, pero no hay más. En este caso la query se trata como en el estado primero y se ofrece una respuesta que se ajusta a la query pero no tiene el grado de perfección del estado 1.

3.1.6. El algoritmo de ranking

El algoritmo de ranking usado se basa en el modelo vectorial explicado en [11]. Nuestra implementación se basa en una implementación previa realizada por Carolina Bonacic [15] en C++ y para la que se han generado los wrappers correspondientes para adaptarla a nuestro modelo de datos y poder usarla desde UPC.

El modelo vectorial se llama así porque las consultas y los documentos se transforman en vectores t-dimensional (donde t es el número de términos de la query) respectivamente de la siguiente manera:

$$\vec{q} = (w_{1,q}, w_{2,q}, ..., w_{t,q})$$

$$\vec{d_j} = (w_{1,j}, w_{2,j}, ..., w_{t,j})$$

Donde $w_{i,j}$ es el peso del termino i en el documento j y $w_{i,q}$ es el peso del término i en la query q. Los pesos $w_{i,j}$ se generan a partir de la frecuencia de aparición del término i en el documento j de la siguiente manera:

$$w_{i,j} = f_{i,j} * \log \frac{N}{n_i}$$

Donde $f_{i,j}$ es la frecuencia absoluta de aparición del término i en el documento j, N es el número de documentos indexados y n_i la cantidad de documentos en el sistema donde aparece el término i.

Una vez tenemos los dos vectores podemos hallar la similitud de la query con el documento de la siguiente manera:

$$sim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| * |\vec{q}|}$$

Que no es más que el coseno que forman ambos vectores. Si la similitud alcanza cierto valor límite se considera que el documento responde satisfactoriamente a a la query. Si ninguno de los documentos obtenidos en una iteración supera dicho umbral el sistema de ranking solicitará otra iteración de K documentos.

3.1.7. Manual de uso

Compilación

En el subdirectorio *codigo* del programa se encuentra el Makefile usado para compilar así como un pequeño script llamado *comp.sh*, que ajusta los parámetros de compilación y lanza el proceso, generando un ejecutable de nombre *main* en el subdirectorio *binarios* del código del programa.

La sintaxis de comp.sh es la siguiente:

comp.sh <tipo de comunicacion><numero de threads><permitir profiling><opciones de optimizacion>

- Tipo de comunicación: puede ser **u** para UDP o **s** para SMP. Originalmente también estaba disponible el modo **m** para MPI pero no fue posible hacer compatible este modo con la necesidad de enlazar UPC con C++
- Número de threads: Un entero positivo que indica el número de threads en el que de dividirá la aplicación.
- Permitir profiling: **p** para permitir el uso de herramientas como PPW o **np** para no permitirlo. El uso de la opcion **p** deshabilita el uso del compilador gccupc pasando a usar el traductor online de berkeley que puede dar problemas con las extensiones BUPC usadas. No se recomienda su uso.
- Opciones de optimización: **debug** para compilar con las opciones -g y –save-temps que fuerzan el modelo de consistencia secuencial y generan información de depuración. **optim** para activar todas las opciones de optimización. Si se omite este parámetro se compilará con el modelo de consistencia relajado y sin optimizaciones.

Uso

El ejecutable recibe los siguientes parámetros:

- Ruta al directorio donde se encuentran los subdirectorios bbdd,consultas y resultados.
- Nombre del fichero de índice.
- Nombre del fichero de consultas.
- Nombre del fichero que asocia palabras a identificadores
- Nombre del fichero que define stopwords
- Nombre del fichero de pesos de palabras
- Un índice de trabajo que se concatena al nombre del fichero de resultados para depuración.

En el directorio *binarios* se adjuntan scripts para ser lanzados desde Sun Grid Engine que muestran el uso de dichos parámetros. Además se adjuntan los directorios de datos con datos válidos preparados para poder probarlos.

3.1.8. Pruebas y resultados experimentales

Objetivos

A la hora de realizar los benchmark y demás pruebas nuestros objetivos fueron los siguientes:

1. Comprobar que el sistema es realmente escalable.
2. Determinar un tamaño óptimo de lote de consultas para deteminar en qué condiciones de tráfico se comporta mejor el sistema.

Entorno de trabajo

Para las pruebas de corrección del sistema se utilizó un cluster heterogéneo con 4 máquinas virtuales montadas sobre VMWare con 512 Mb de RAM cada una y Kubuntu 8.04 como sistema operativo. Por otra parte las pruebas reales de rendimiento se realizaron sobre el cluster ETNA del grupo ArTeCS que posee 8 nodos v20z destinados a cómputo y otro para frontend con las siguientes características:

- 2 x AMD Opteron Dual Core 270 a 2Ghz
- 4 Gb de Memoria RAM

- 1 x HDD 73Gb Ultra320 10K

- 2 x 10/100/1000 Ethernet

El almacenamiento se realiza sobre un array de 12 discos SCSI de 146Gb conectados al frontend y las comunicaciones de datos y de gestión se realizan sobre redes Gigabit Ethernet independientes.

Primeras pruebas

Para satisfacer los objetivos de las pruebas se prepararon varios lotes de pruebas de ejecución. Estas pruebas consistieron en la ejecución diez veces del programa con cada juego de parámetros. Los parámetros con los que se trabajó son los siguientes:

- Número de threads: 1,2,4,8,16

- Tamaño del lote de consultas: 1000,2000,8000

- Tamaño de la iteración: 64,128

Para cada conjunto de parámetros se extrajo el throughput máximo del sistema. Los resultados obtenidos han sido los siguientes (en consultas/s):

Tamaño del lote / Nº de nodos	1000	2000	8000
1	3023,3	3123,7	3071,4
2	2855,1	2171,1	2590
4	4692,7	4215,7	4781
8	7894,5	7973,7	7199,8
16	9480,3	9341,9	9154

Cuadro 3.1: Resultados para K=64

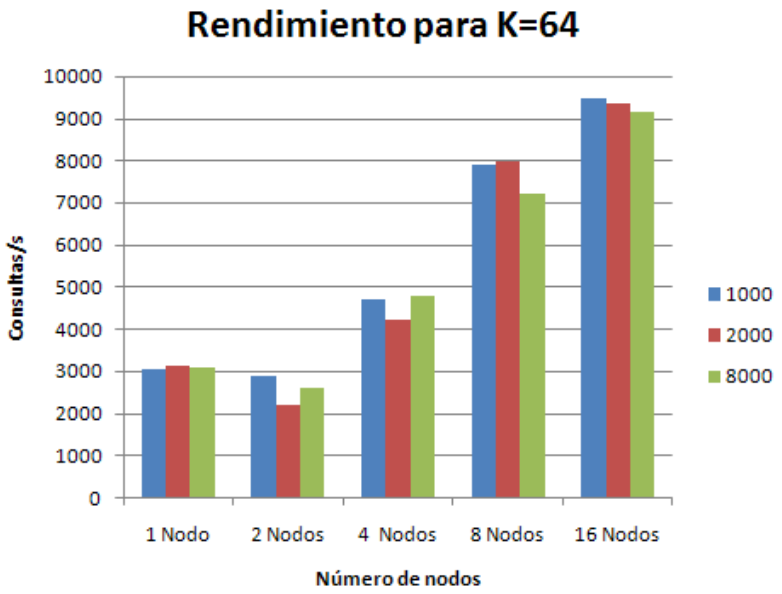

Figura 3.4: Resultados en solo lectura para K=64

Nº de nodos \ Tamaño del lote	1000	2000	8000
1	3124,9	3074	2910,1
2	2857,5	2751,2	2802,1
4	4592,7	5322,4	4404
8	8691,	7899,4	7427,5
16	11823	11290	11188

Cuadro 3.2: Resultados para K=128

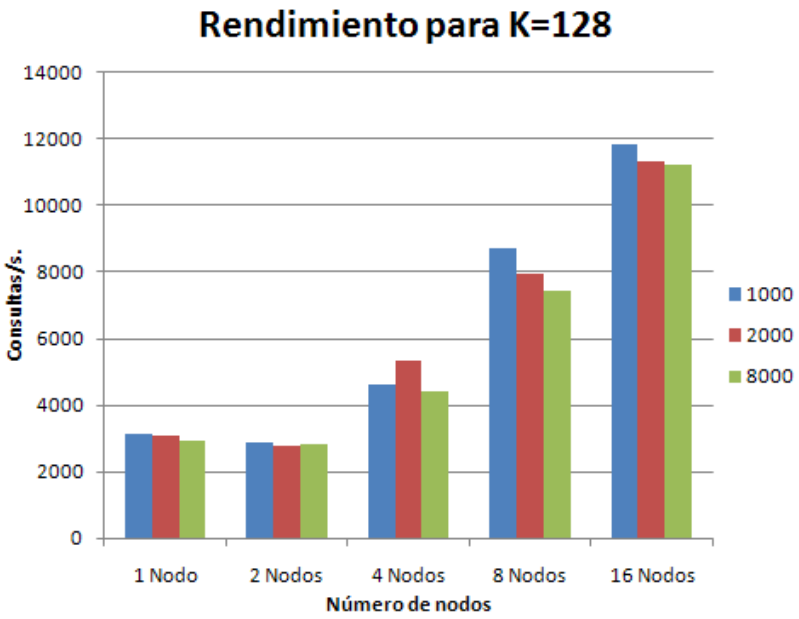

Figura 3.5: Resultados en solo lectura para K=128

De estos resultados hemos extraido las siguientes conclusiones:

- Para tamaños de iteración K=128 los resultados son mejores. La mejoría no es tan grande como podría esperarse ya que nuestro índice invertido contiene pocos términos que existan en más de 64 documentos y por tanto la mayoría de querys se resuelven en 1 iteración para ambos valores de K

- El tamaño de lote más adecuado parece estar en torno a 1000. Más adelante realizaremos una pequeña prueba para intentar acotar mejor este valor.

- El sistema efectivamente es escalable.

- Hay una anomalía en el paso de 1 thread a 2. A pesar de que 2 threads deberían comportarse mejor en las pruebas se comportan de forma consistente de forma peor. Hemos especulado con las posibles causas de que el paso de una sola máquina a dos provoque esa pérdida de rendimiento (a priori se le ha achacado a la necesidad de comunicación entre máquinas,o un posible exceso de llamadas al sistema) pero no ha sido posible llegar a una respuesta satisfactoria.

A continuación presentamos la prueba realizada para intentar acotar mejor el tamaño ideal de lote. Para ello se han realizado pruebas con la siguiente configuración: K=128, Número de nodos=8 , Tamaño de lote=100,500,750. Los resultados son los siguientes:

Tamaño del lote	100	500	750	1000	2000	8000
	7405	7728,7	7699	8691,4	7899,4	7427,5

Cuadro 3.3: Resultados de ajuste de tamaño de lote

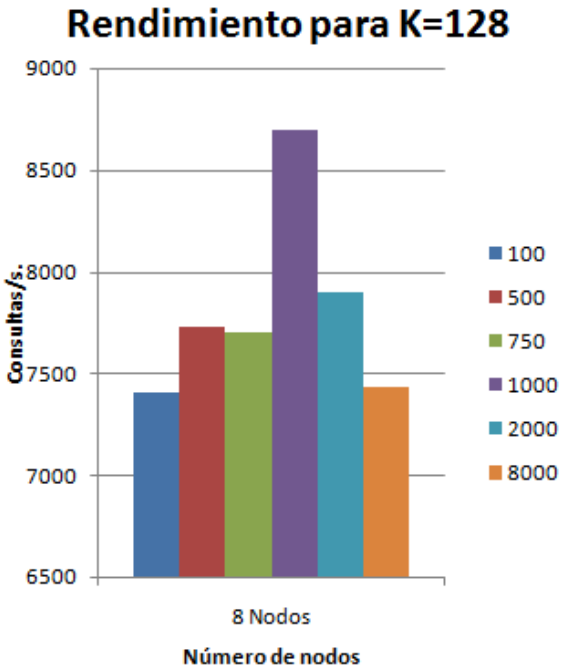

Figura 3.6: Estudio de productividad segun el tamaño de lote

Los resultados indican que el tamaño de lote óptimo para nuestro sistema se encuentra efectivamente en 1000 resultando sensiblemente inferiores para valores por encima y por debajo de dicho valor.

Finalmente realizamos una comparativa de rendimiento entre esta versión sin opciones de optimización en compilación y la misma versión compilada con -O3 para los ficheros de C++ y con -opt-enable=split-phase,pre-add,ptr-coalesce,ptr-locality,forall-opt. No se ha activado ninguna optimización de UPC que aun se encuentre en fase experimental. Los siguientes resultados muestran una mejora muy significativa del rendimiento pero siguen manteniendo el comportamiento extraño en el paso de uno a dos nodos.

Número de nodos	1	2	4	8	16
	6288	5514	9693	15984	19180

Cuadro 3.4: Resultados para K=128 con optimizaciones

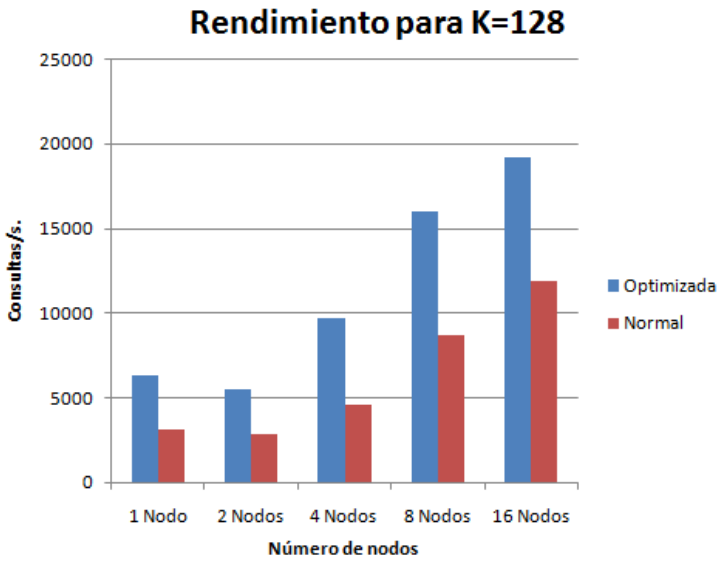

Figura 3.7: Comparativa de productividad entre versiones con/sin optimizacion

3.2. Operaciones de lectura con escrituras online poco frecuentes

3.2.1. Introducción

Nuestro objetivo en este caso es ampliar el ranker de solo lectura permitiendo poder hacer operaciones de escritura online. Se considerará que las operaciones de escritura son muy poco frecuentes (poner una noticia de ultima hora por ejemplo) y que el grueso de las actualizaciones se sigue realizando offline.

Debido a restricciones de tiempo no existe una implementación funcional de esta versión. Por tanto comentaremos el diseño ideado para el problema y los motivos por los que se ha decidido hacerlo de esa manera.

3.2.2. Distribución de los datos

El modelo de datos de la versión 7 estaba optimizado para lecturas y es muy poco eficiente para considerar escrituras online. Por eso hemos diseñado el siguiente modelo:

Posting list

La representación en array compartido supone un coste lineal en el número de documentos del sistema cada vez que se quiera añadir un nuevo documento a la base de datos. Aunque las escrituras sean poco frecuentes, un coste de esta magnitud es totalmente inaceptable y requiere de cambios profundos en la manera de organizar los datos.

El requisito de distribuir los K documentos por iteración entre los nodos de proceso sigue vigente. La adición de nuevos documentos no debe suponer un coste excesivo (podemos permitirnos más coste que en lecturas pero ni mucho menos el coste lineal que tenía la versión anterior). Usar una lista enlazada simple tampoco era factible: el coste de los accesos remotos para ir siguiendo los punteros resulta prohibitivo. Se decidió pues que necesitabamos una estructura que:

1. Permitiera obtener todos los datos de una iteración con una sola llamada a las funciones de obtencion de datos de las extensiones BUPC.

2. Contuviera en sí misma los punteros necesarios para acceder a la siguiente estructura.

3. Fuera igual de eficiente para lecturas y no penalizara en exceso las escrituras.

El hecho de que el algoritmo vectorial de ranking dé los mismos resultados aunque se varíe el número de documentos que se le suministran por iteración permite que la estructura pueda tener espacio libre extra que puede ser utilizado para inserciones rápidas. Cuando ese espacio libre extra se llena se puede proceder a reestructurar esa unidad de informacion junto con las inmediatamente adyancentes, teniendo así penalizaciones muy pequeñas en escritura salvo una un poco más costosa cuando se llena el espacio extra.

Otra ventaja reside en que los datos dentro de una misma iteración no tienen por qué llevar ningun tipo de orden entre ellos. Sólo es necesario garantizar que todos los documentos de un término para una iteración I tienen una frecuencia de aparicíon mayor que la de la iteración I+1. Por tanto las inserciones de nuevos documentos se pueden realizar de manera secuencial en el espacio extra sin preocuparse de reordenar. Para poder realizar esto cada bloque debe contener los valores de frecuencia máxima y frecuencia mínima que se deben actualizar en escrituras.

Teniendo todo esto en cuenta se planteó la estructura de las figuras 3.8 3.9 (optimizada para una estructura de 64 bit):

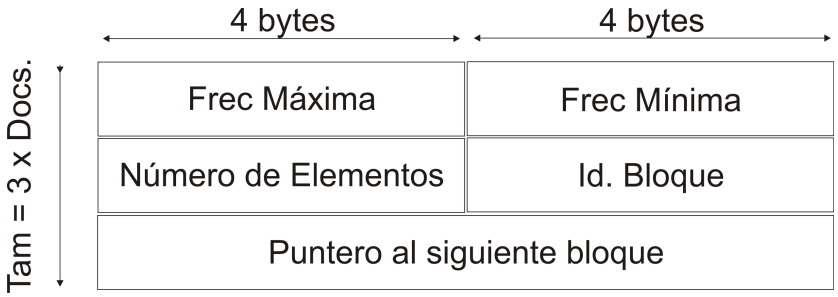

Figura 3.8: Cabecera de bloque de documentos

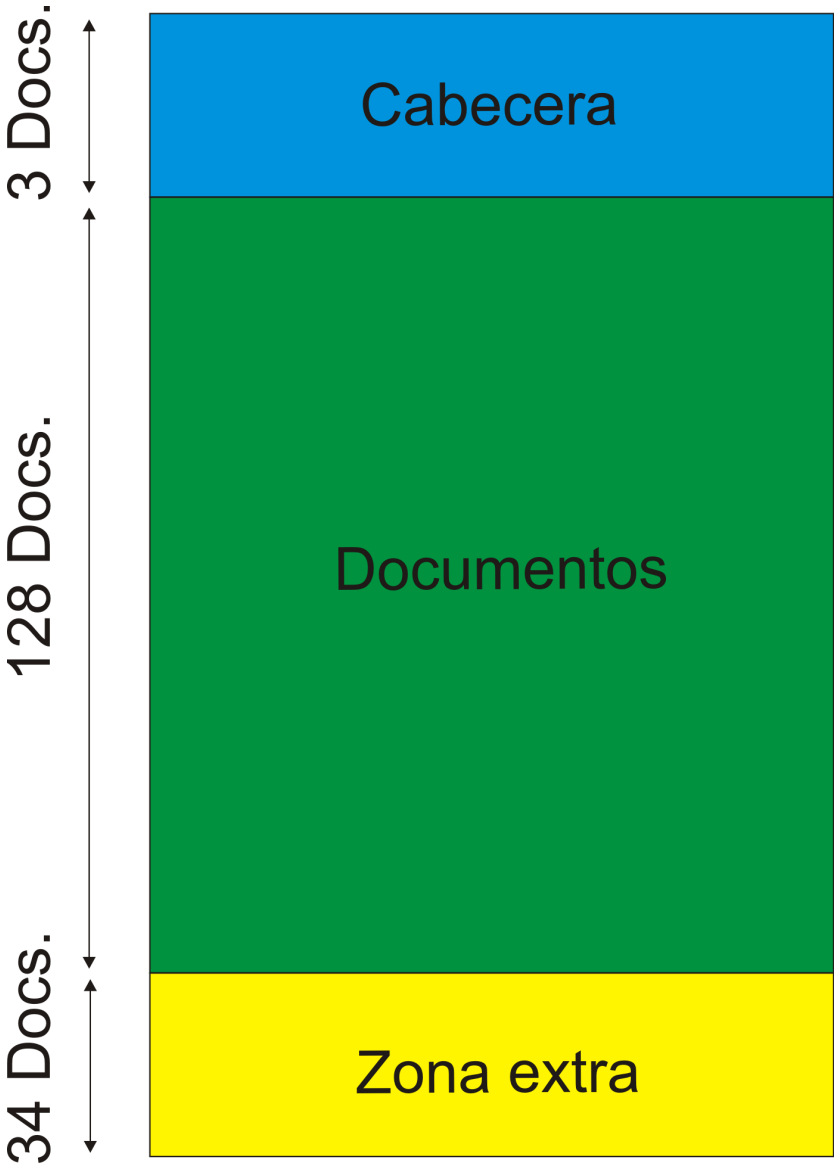

Figura 3.9: Bloque de documentos en la versión con escritura

Cada bloque de documentos se ubica en memoria compartida con un tamaño de round-robin igual al tamaño de la cabecera, lo que hace cómodo su uso.
Con una sola operación BUPC de obtención del bloque desde memoria compartida conseguimos por tanto:

- Un puntero al siguiente bloque de datos. Al obtener el puntero junto con los documentos reducimos acccesos a memoria compartida y podemos comprobar si hay datos disponibles para la siguiente iteración.
- Las frecuencias máximas y mínimas de los documentos del bloque para poder soportar el modelo de escrituras en la zona extra.
- El número de documentos real del bloque y un identificador de bloque (éste último está por motivos de depuración y para poder alinear correctamente los campos de la cabecera en un espacio de 3 documentos).
- Una cantidad de documentos entre 128 y 128+34 que conforman todos los necesarios para una iteración.

Lista de términos

Puesto que en la versión con escritura se ha planteado poder añadir nuevos términos online hemos optado por cambiar el array de términos por un vector de la librería STL de C++. Cada entrada de este vector se corresponde con la siguiente estructura:

```
struct Termino
{
        int idTermino;
        shared BloqueDocumentos* primerBloque;
}
```

Que únicamente contiene el identificador del término y un puntero para acceder al primer bloque de datos del mismo.

3.2.3. Operaciones

Obtención de datos

El funcionamiento a nivel de lecturas es idéntico al de la versión anterior. La única diferencia consiste en un pequeño preprocesado del bloque obtenido para obtener el número de documentos, determinar si va a haber más datos para futuras iteraciones y pasar al ranker sólamente los documentos sin cabecera. Como se ha comentado en secciones anteriores no supone ningun problema que la cantidad de documentos por

iteración varie ligeramente, por lo tanto se le pasan todos los documentos obtenidos en el bloque.

Escrituras

La inserción de un documento implica los siguientes pasos:

- Determinar el bloque de datos donde debería ir el nuevo documento. Esta operación puede realizarla de forma eficiente en Thread 0 ya que las cabeceras por el reparto round robin siempre tienen afinidad por ese Thread. Consistiría en una búsqueda lineal bloque a bloque comprobando las frecuencias mínimas y máximas. La operación efectiva de actualización puede ser realizada por el Thread 0 o por el thread que tiene afinidad por la posición de escritura (se le comunica por el sistema de mensajes integrado).
- Intentar una inserción en el bloque. Si hay espacio en el bloque se inserta en el primer hueco libre (la posición libre se obtiene en tiempo constante a partir del número de documentos del bloque) y la operación ha terminado. Si no hay hueco se procede a crear un nuevo bloque cuyo contenido será parte del bloque actual y de los bloques anterior y posterior y se procede al reordenado de los bloques afectados y a la actualización de cabeceras.

El proceso de escritura precisa del uso de sistemas de sincronización. Una parte es facilmente obtenible por el sistema de mensajería del programa. Para la otra nos planteamos usar los semáforos de las extensiones BUPC que dan un rendimiento entre 2 y 2.5 veces mayor que los cerrojos de upc [10].
En la fase de diseño se han planteado diversas posibilidades respecto al nivel de granularidad de la sincronización de escritura:

- De grano muy fino: bloquear las lecturas a nivel de término. Ésta solucion fue descartada de forma casi directa por el inmenso número de semáforos necesarios (Núm Terms*Threads).
- Granularidad por conjunto de términos : considerar los términos agrupados según alguna función hash y bloquear las lecturas sólo de los grupos que están siendo actualizados.
- De grano grueso: Si se está realizando una escritura se bloquean todas las lecturas. Es de lejos la más ineficiente y sólo se planteaba su uso para depurar las operaciones de escritura.

Se decidió usar la versión de granularidad por conjuntos pero por limitaciones de tiempo no se pudo realizar una implementación funcional ni las pruebas necesarias para demostrar su rendimiento.

Conclusiones

Capítulo 4

Conclusiones

4.1. Sobre la implementación del buscador

Los resultados de las pruebas realizadas con nuestra implementación del motor de búsqueda son bastante positivos. La implementación ofrece un buen rendimiento y es escalable hasta 16 nodos (probablemente aun sea posible aumentar ligeramente el número de nodos pero la falta de medios técnicos nos ha impedido realizar pruebas con un número de nodos superior).

Sin embargo en el momento de escribir estas conclusiones quedan pendientes dos aspectos fundamentales que podrían ser motivos de trabajos futuros:

- Investigar en profundidad los motivos por los que se produce una pérdida de rendimiento considerable cuando pasamos de uno a dos nodos. Aunque se han explicado previamente algunas hipótesis al respecto, sería conveniente una investigación más en profundidad que permitiera acotar exactamente cual es el problema. Es posible que si se descubrieran las causas se pudiera obtener una escalabilidad superior.
- Completar la implementación de la versión con escrituras. Por falta de tiempo no se ha podido realizar una implementación completa, pero no debería resultar demasiado dificil transladar las ideas expresadas en la sección 3.2 a una implementación concreta para poder comprobar la validez de los planteamientos diseñados.

4.2. Sobre el lenguaje UPC

Las pruebas han demostrado que UPC permite compilar programas que escalan bastante bien, pero durante el desarrollo nos hemos encontrado con bastantes dificultades [1]:

[1] Muchas de ellas probablemente asociadas a la versión libre del compilador que usábamos

- Falta de documentación: Aunque existen manuales,referencias de lenguaje y ejemplos de UPC muchos de ellos no están actualizados desde hace años y es muy dificil encontrar información sobre problemas concretos. Como ejemplo concreto podemos hablar de las extensiones BUPC que aparecen en [10] y [9] y que aparecen en no más de tres resultados en búsquedas en google. Estas extensiones están implementadas pero sólo aparecen documentadas en un par de papers que no son más que proposiciones (y de hecho la implementación varía ligeramente de la presentada en los documentos).

- Problemas con el modelo de punteros compartidos: Las limitaciones de tamaño de bloque en los punteros compartidos empaquetados no han permitido ejecutar problemas excesivamente grandes en algunas de nuestras pruebas. Aunque existen maneras teóricas de configurar el runtime de Berkely junto con el compilador GCCUPC para adaptar el uso de punteros a nuestras necesidades, estas no suelen funcionar (no están implementadas o no son tenidas en cuenta por el configurador de compilación), lo cual nos vuelve a llevar al asunto de la documentación desfasada.

- Fuertes penalizaciones de rendimiento en accesos a datos remotos de forma individual: aunque se esperaban bajadas de rendiminto en estos casos, las penalizaciones han sido mucho más fuertes de lo esperadas. Las opciones de optimización en compilación no parecen tener en cuenta accesos secuenciales a matrices remotas y no parecen ofrecer mejoras de rendimiento significativas (de hecho casi todas las optimizaciones con cierta importancia se obtuvieron a nivel de compilación en C++ y no en UPC). Esto nos ha forzado a intentar optimizar a mano todos los accesos a memoria, limitando nuestra productividad a la hora de implementar las distintas versiones.

Aun así tambien nos han sorprendido gratamente algunos aspectos del compilador de UPC que usábamos:

- La posibilidad que ofrece GCCUPC + Berkeley Runtime de elegir distintas capas de transporte de datos ofrece mucha versatilidad y permite obtener en algunos casos mejoras de rendimiento significativas usando protocolos de comunicación ligeros como son UDP sobre otros más pesados como MPI.

- La sintaxis del lenguaje que permite acceder a datos remotos como si fueran locales hace de UPC un lenguaje con una expresividad superior a otros lenguajes orientados a programación paralela. Si los diseñadores del lenguaje implementaran operaciones de transferencia de datos que pudieran abstraer al programador en determinados casos de la distribución fisica de la memoria, UPC podría convertirse en un lenguaje muy cómodo de usar y con una buena productividad.

Parte IV

Apéndices

Configuración de un cluster virtual

.1. Motivación

Para poder realizar correctamente las tareas de investigación sobre el funcionamiento y configuración de UPC se necesitaba acceso de root, lo cual obviamente no estaba disponible en el entorno de trabajo. Además las tareas de depuración de nuestro software no debían quitar tiempo de procesamiento en ETNA a otros programas que sí estan terminados y necesitan todo la capacidad de procesamiento disponible.
Por eso decidimos aprovechar toda la infraestructura disponible (tres ordenadores portátiles) y montar un cluster de máquinas virtuales que nos permitiera trabajar con tranquilidad.
A continuación mostramos como crear 2 máquinas virtuales como ejemplo:

.2. Manual de configuración

Para crear el cluster virtual, utilizaremos el programa VMware 6.0. En este apéndice se utilizaran dos nodos de ejemplo. Uno actuará como nodo maestro (shark), mientras que el otro actuará como nodo esclavo (remora). Para aumentar el tamaño del cluster tan solo hay que replicar más nodos esclavos y ajustar su configuración para enlazarlos correctamente al nodo maestro. Los detalles de configuración se describen a continuación.
Como sistema operativo para las máquinas virtuales utilizaremos una distribución de Linux ya que los programas utilizados están optimizados para sistemas UNIX.

Características básicas de las máquinas virtuales

Las máquinas virtuales han de asentarse sobre equipos físicos de características parejas para que el rendimiento de cada máquina virtual no se vea afectado.
A continuación se detalla la configuración general de las máquinas virtuales a la hora de crearlas con VMware.

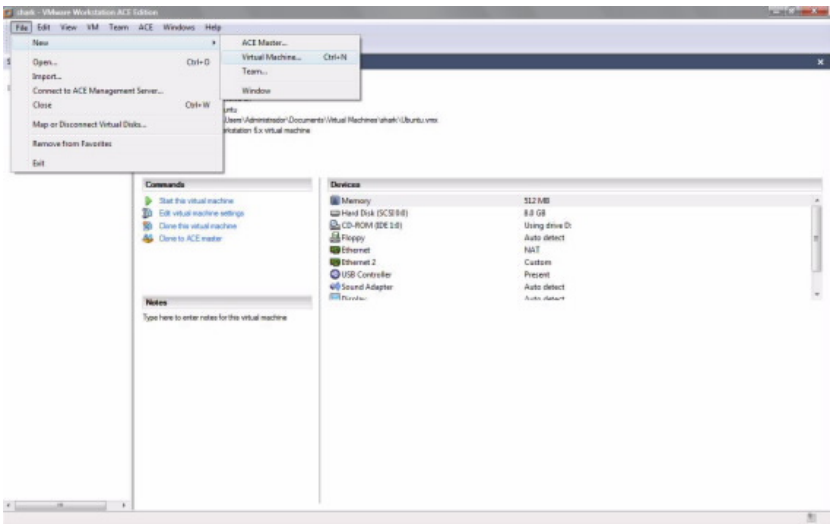

Figura 1: Crear una nueva máquina virtual

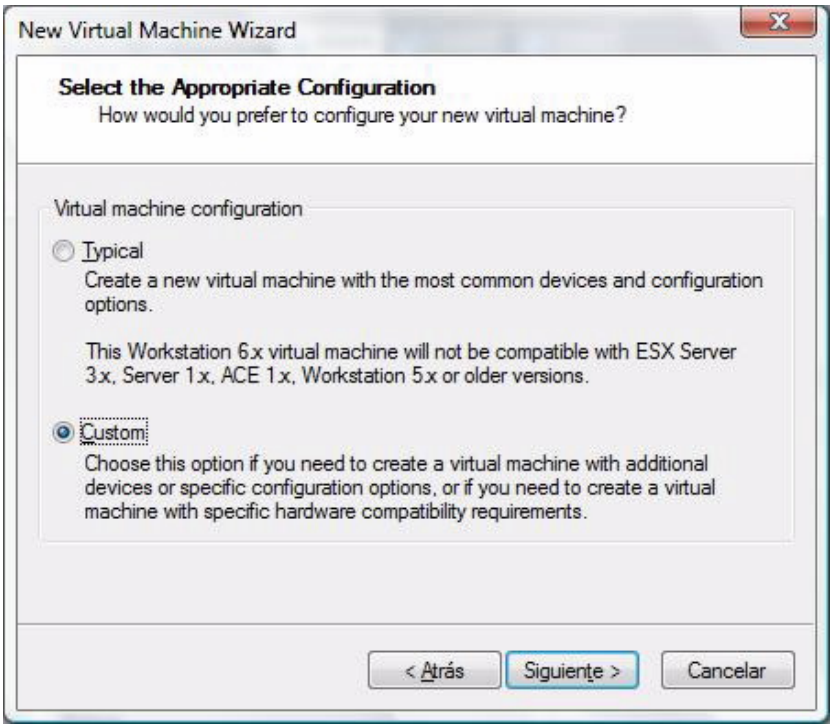

Figura 2: Selección del tipo de VM

Elegimos la opción personalizada. De esta forma ajustaremos la configuración a las especificaciones de la maquina que nos dará soporte.

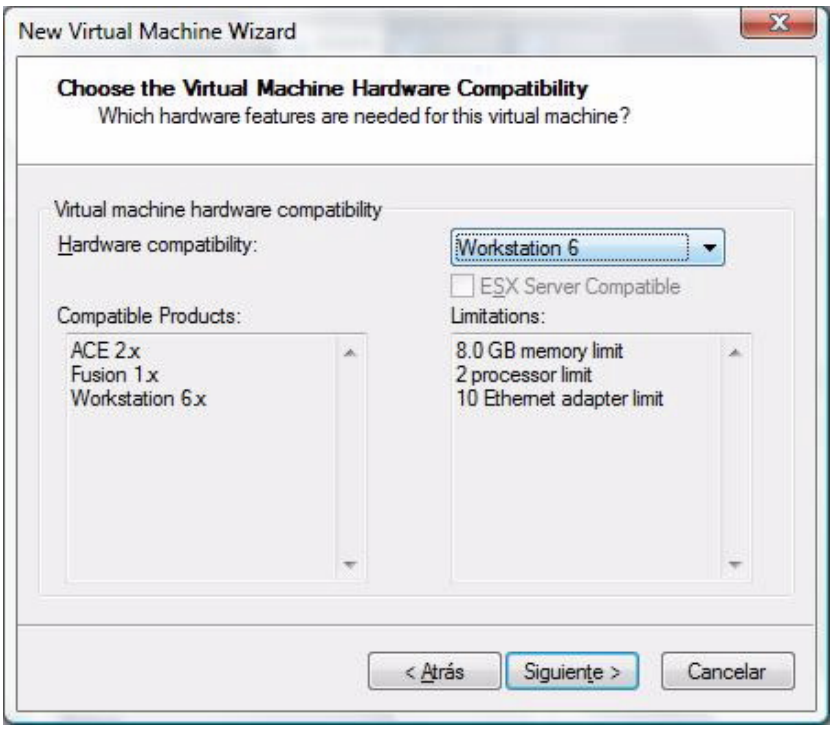

Figura 3: Selección compatibilidad de hardware de la VM

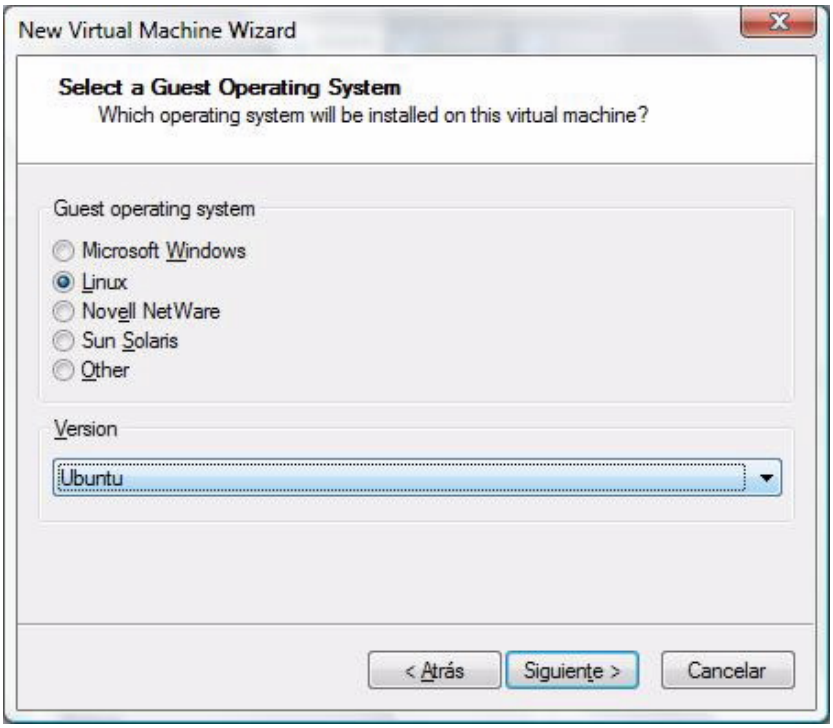

Figura 4: Selección guest OS de la VM

Marcamos la opción Linux, y como versión Ubuntu, ya que la distribución linux que

utilizaremos es una Kubuntu 8.04.2 (Hardy Heron).

http://releases.ubuntu.com/kubuntu/hardy/kubuntu-8.04.2-desktop-i386.iso

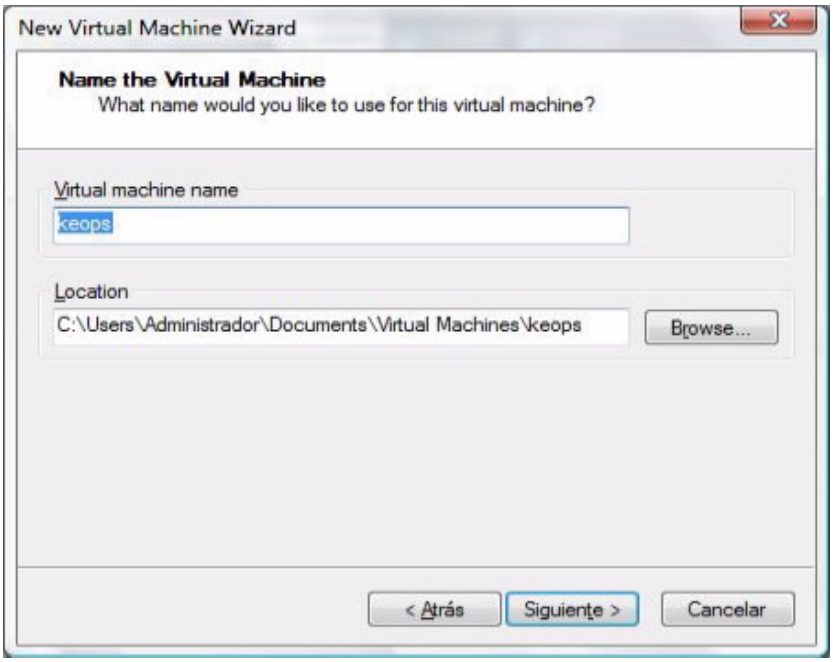

Figura 5: Selección de nombre y ruta de la VM

Damos el nombre que queramos y una ubicación para los archivos de la que necesita VMware para configurar y mantener el estado de la máquina virtual.

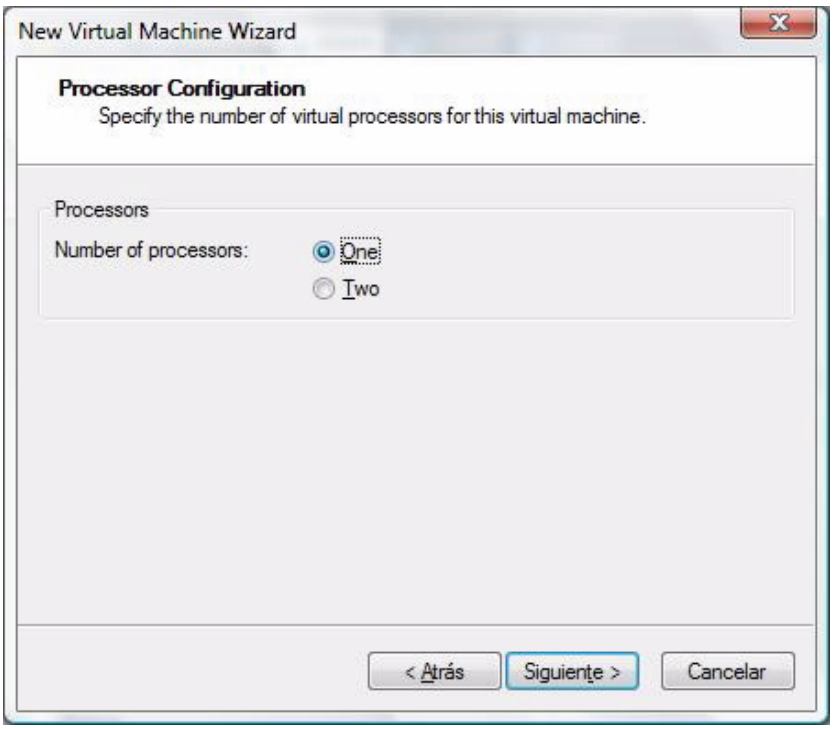

Figura 6: Selección del número de procesadores de la VM

Es importante elegir un solo procesador para evitar que la máquina se sature a causa del trabajo producido por el VMWare.

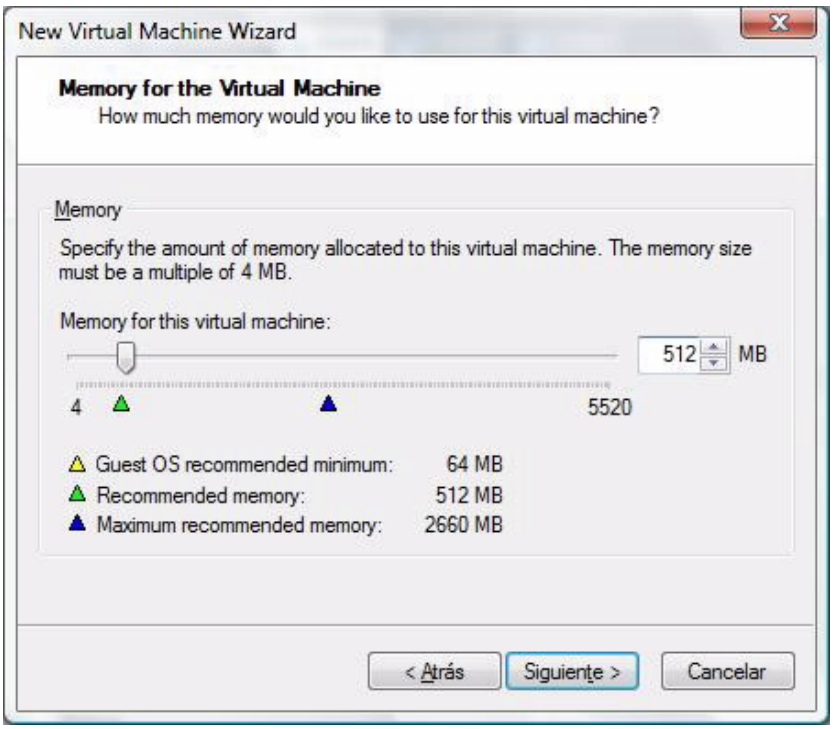

Figura 7: Selección de la RAM disponible para la VM

No utilizaremos mucha memoria RAM en las máquinas virtuales para evitar sobrecargar el sistema. Con la cantidad recomendada (512 MB), podemos mantener dos máquinas virtuales simultáneas. De este modo incluso podemos mantener un entorno gráfico.

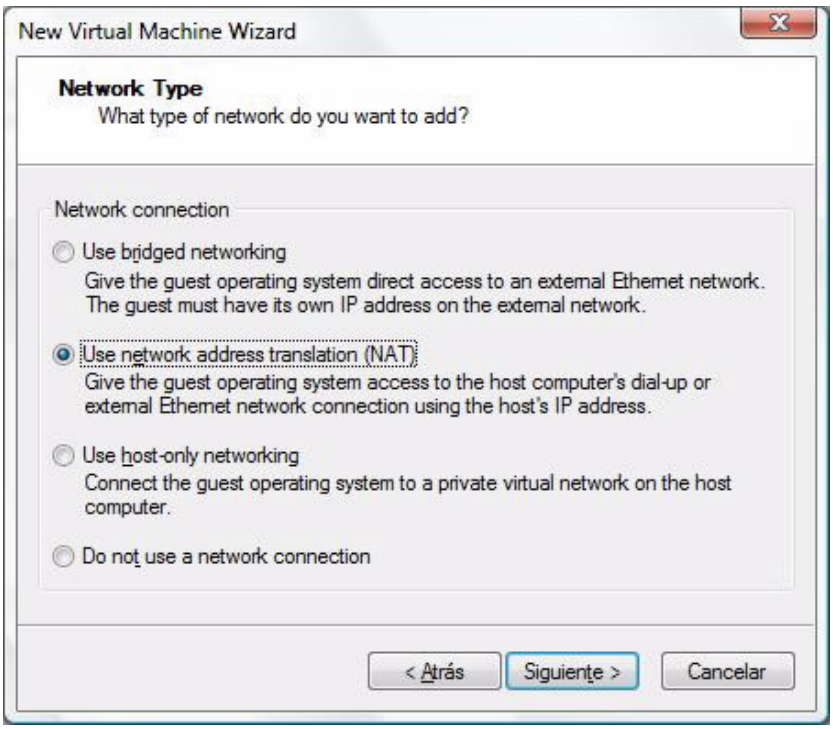

Figura 8: Selección de la red usada por la VM

Elegimos el tipo de conexión NAT para que la máquina virtual tenga acceso a Internet.

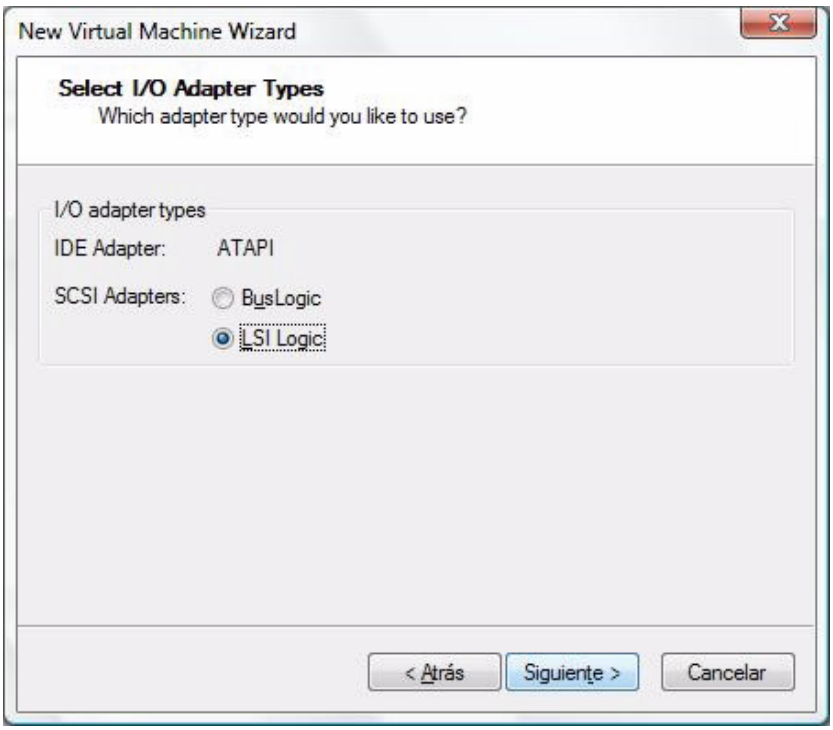

Figura 9: Selección de interfaz de almacenamiento para VM

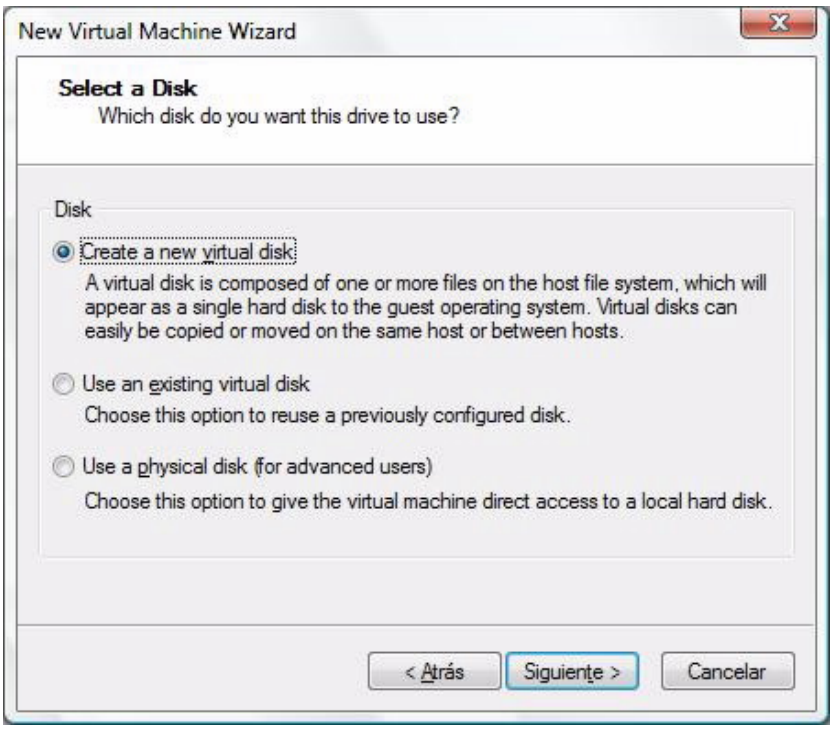

Figura 10: Selección de disco duro para la VM

La opción de disco virtual nos permitirá copiar o mover una máquina virtual de tal forma que facilitará el proceso de clonación de máquinas a la hora de incrementar el número de nodos del cluster.

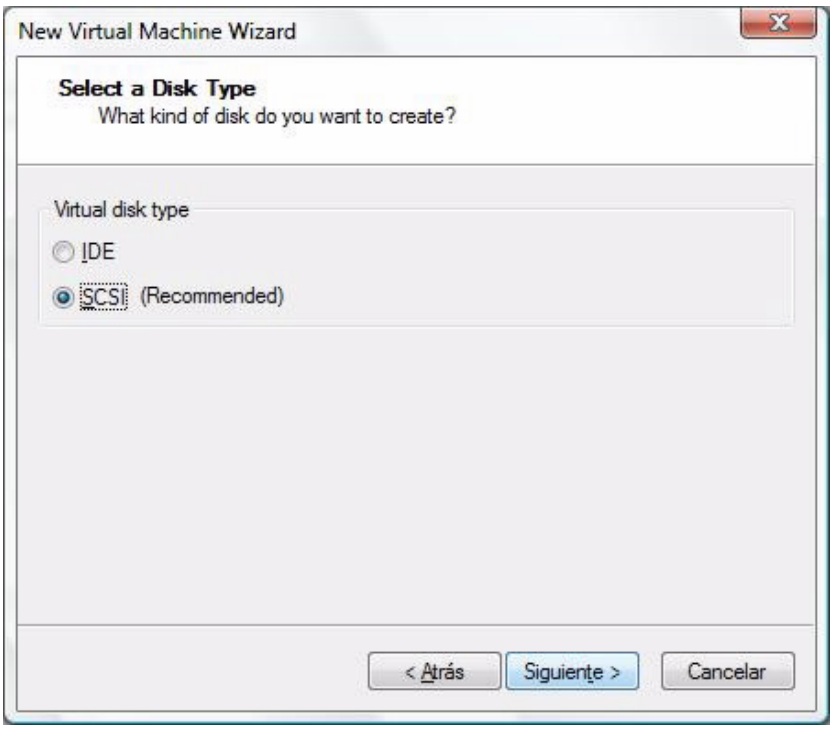

Figura 11: Selección de tipo de disco para la VM

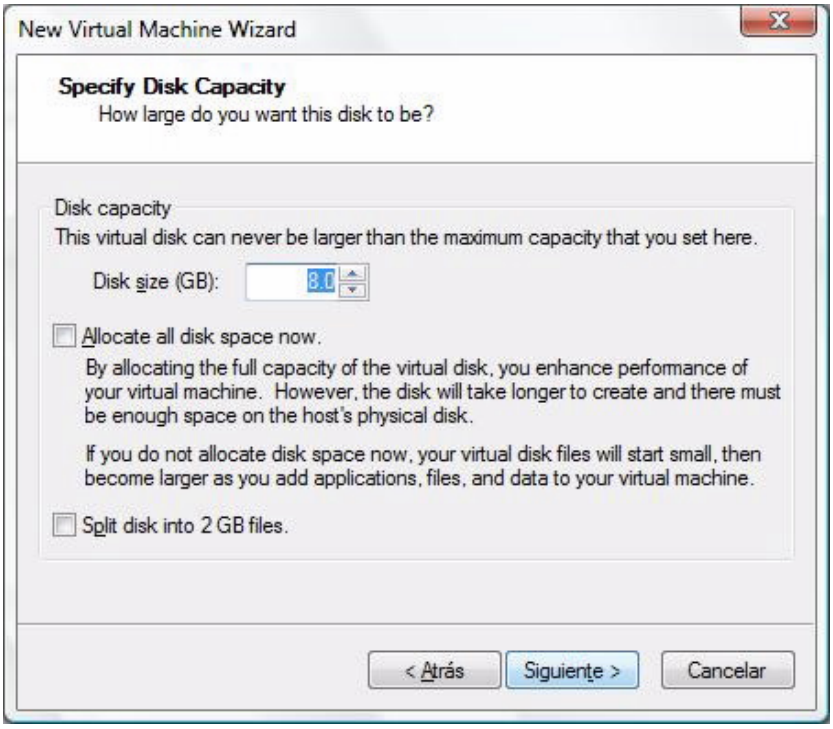

Figura 12: Selección capacidad de disco en una VM

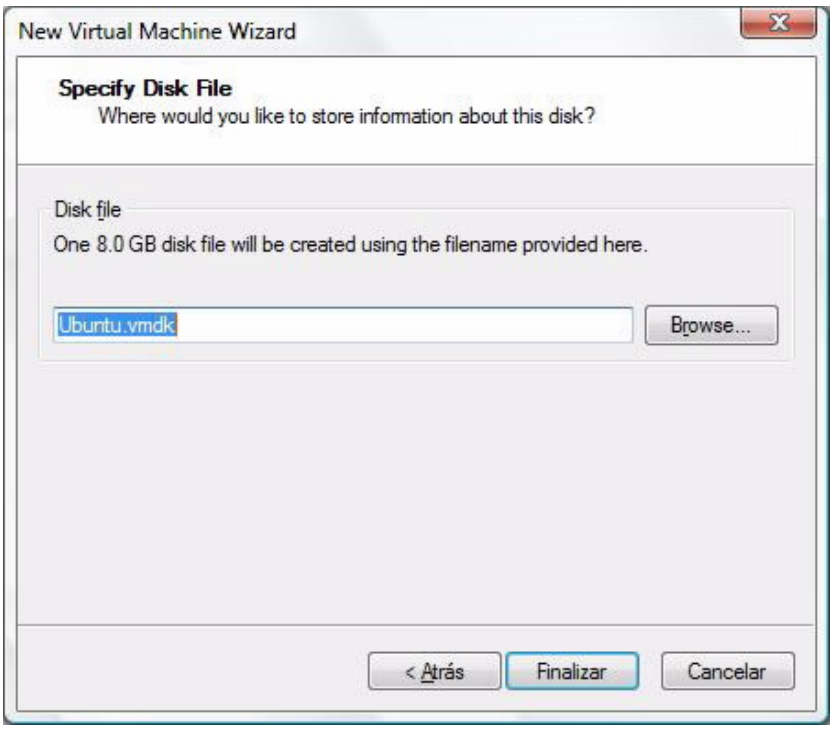

Figura 13: Selección de ruta para disco virtual

Una vez terminada la configuración deberemos instalar el sistema operativo en cada uno de los nodos del cluster virtual.

Configuración de red de los nodos del cluster virtual

La red del cluster virtual corresponde al siguiente esquema:

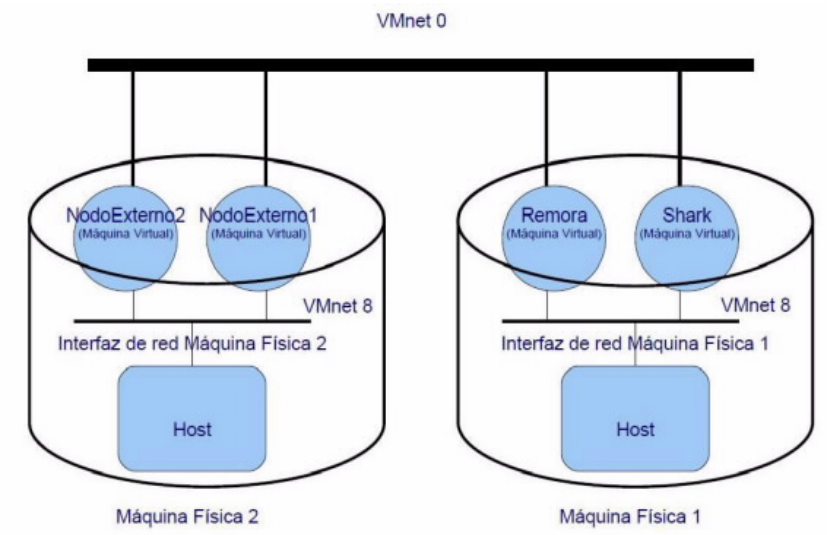

Figura 14: Vision de la red del cluster virtual

La red VMnet 8 es de tipo NAT, que es la conexión que elegimos en la configuración básica de las máquinas virtuales, mientras que la red VMnet 0 es puenteada. Deberemos declarar en la máquina virtual un nuevo interfaz de red e indicarle el tipo de conexión "bridged".

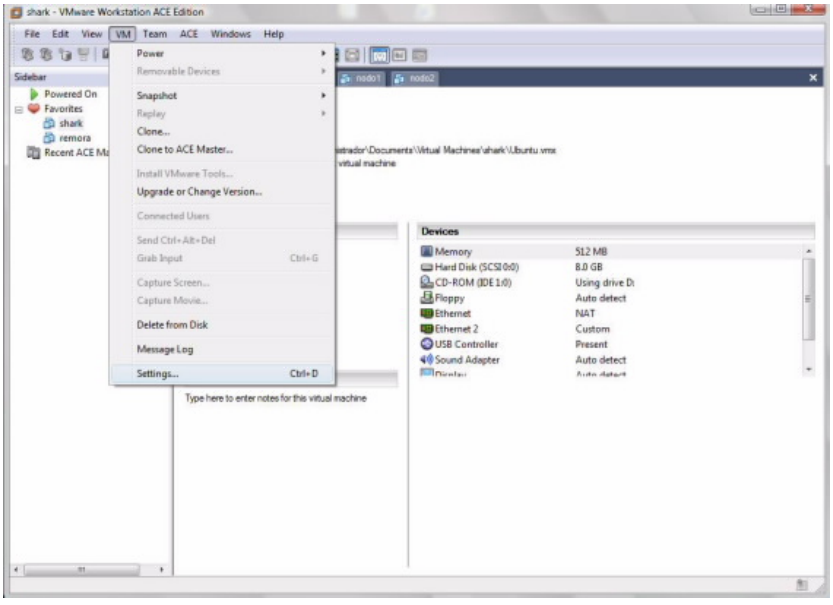

Figura 15: Menú de configuracion de una VM

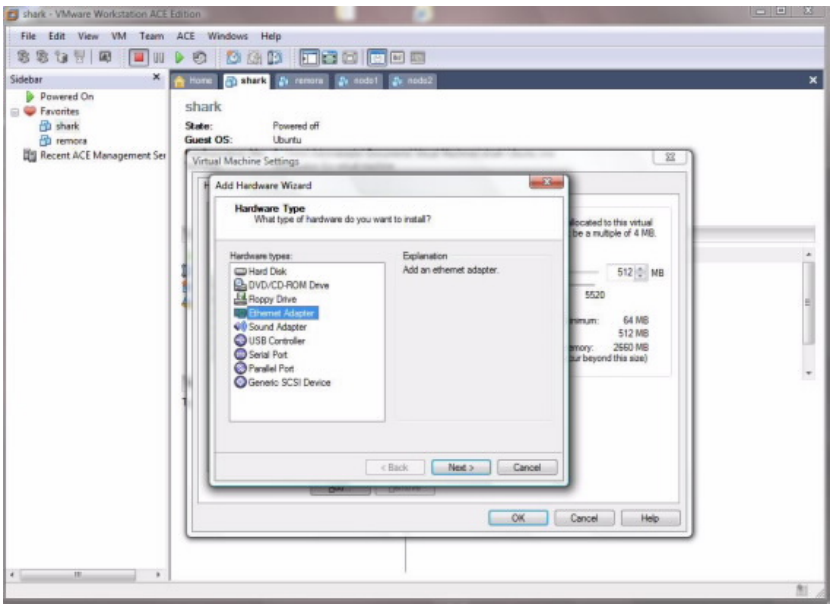

Figura 16: Creacion de un nuevo interfaz de red

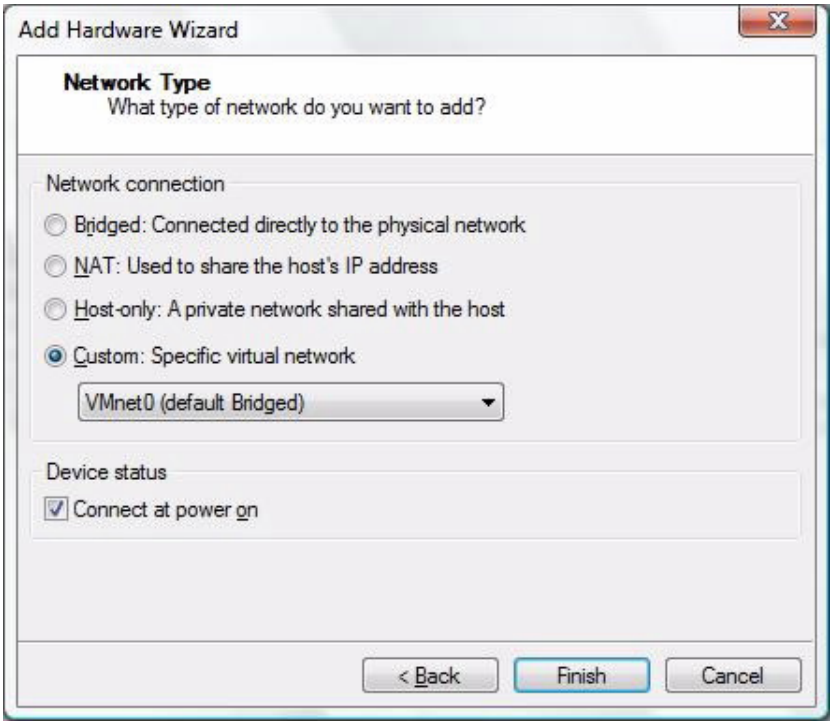

Figura 17: Selección de propiedades del interfaz de red

El siguiente paso es ajustar las direcciones IP de los interfaces del sistema operativo de las máquinas virtuales.

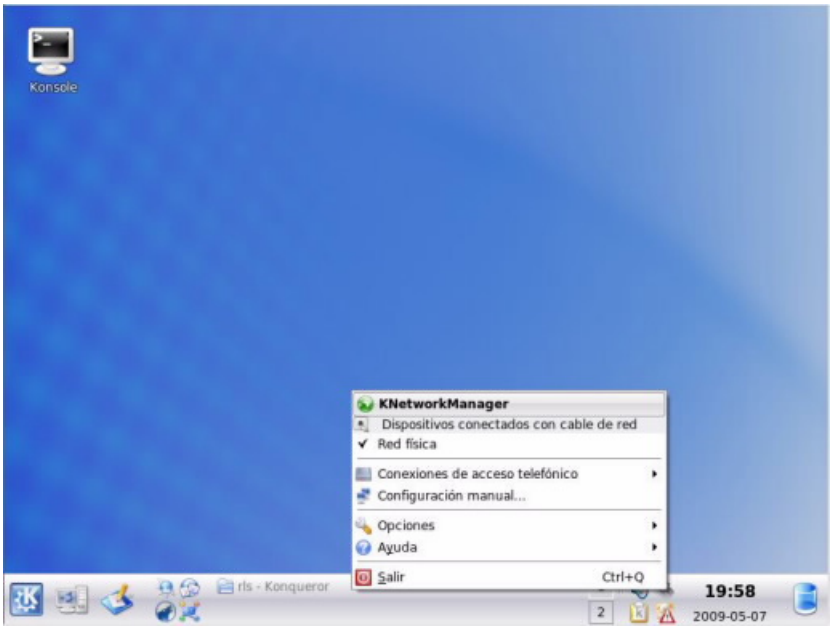

Figura 18: El menu de selección de KNetworkManager

Utilizando el KNetworkManager podremos asignar direcciones IP a cada uno de los interfaces de red definidos y mantener dicha dirección. En este caso"eth0" esta conectado a la red VMnet 8 (de tipo NAT) y el interfaz eth1 a la red VMnet 0 (puenteada), no obstante, es recomendable comprobar que interfaz esta relacionado con que red. Al interfaz "eth1" en este caso, le hemos asignado una red privada de clase A.

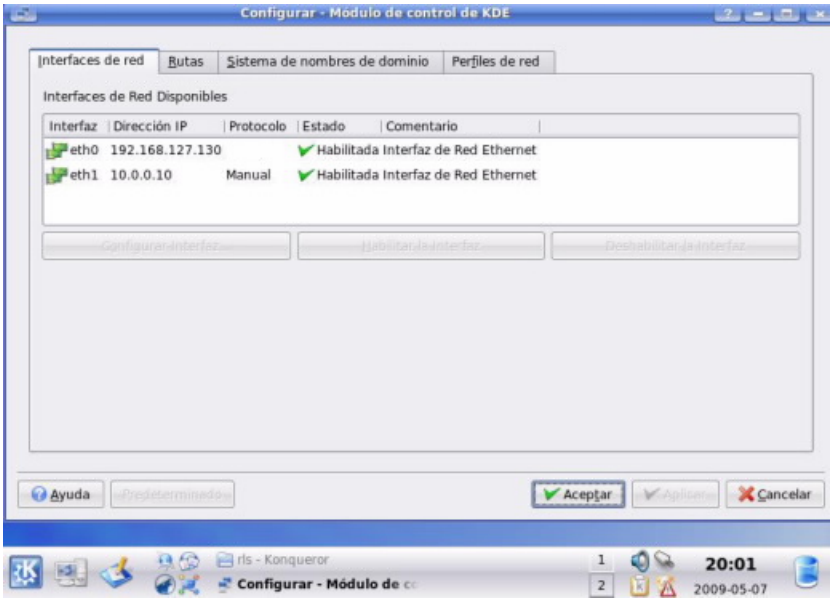

Figura 19: Configurando los adaptadores en linux

Para comprobar que la configuración es correcta, se debería obtener comunicación utilizando el comando “ping”, de tal forma que tanto una red del exterior como otro nodo del cluster respondieran a dicho comando.

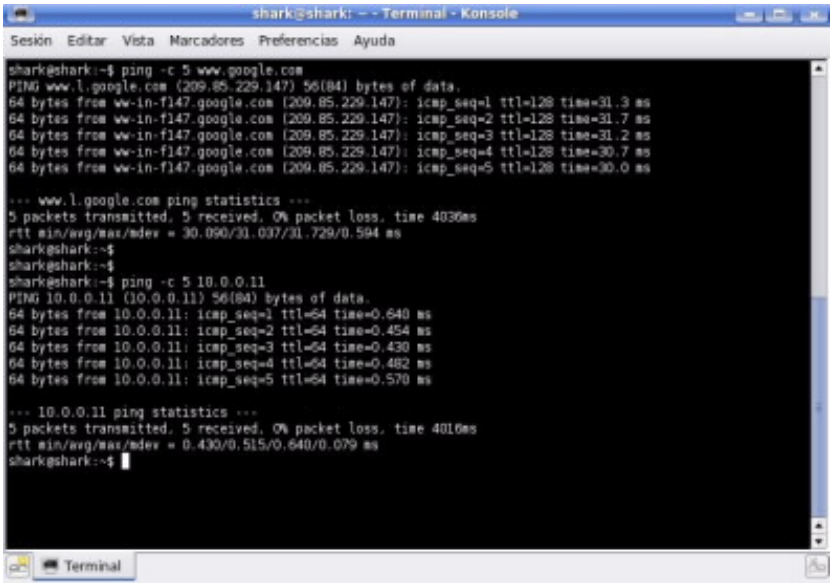

Figura 20: Comprobación de la conectividad en el cluster

Como podemos ver, al hacer ping al domino www.google.com, estamos utilizando la red NAT, mientras que la dirección 10.0.0.11 (remora) estamos usando la red puenteada. Para finalizar editaremos el fichero “/etc/hosts” para conseguir:

- Que cada nodo del cluster virtual conozca a los demás nodos.

- Asociar a las direcciones de los nodos del cluster virtual un nombre.

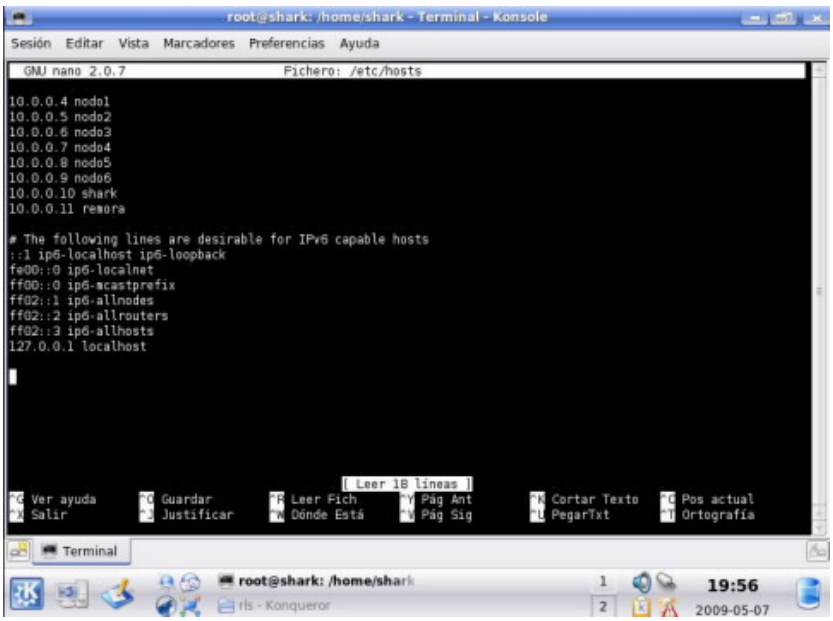

Figura 21: Configuración de los nombres de los equipos

Este archivo tiene que ser editado en cada nodo del cluster virtual.

Configuración e instalación del entorno de trabajo del cluster virtual.

En este apéndice se describen los programas necesarios para obtener un entorno paralelo que actúe como sistema para el desarrollo y la ejecución de programas como el descrito en este proyecto.

Como soporte físico, el entorno de trabajo utiliza un cluster de máquinas virtuales. Para estos apéndices se utiliza como modelo el cluster virtual descrito anteriormente en los apéndices, no obstante, los parámetros e indicaciones utilizadas pueden ser fácilmente extrapoladas a otros sistemas de similares características.

.2.1. Compartir archivos y aplicaciones

El entorno de trabajo necesita que el archivo que se este ejecutando en el base runtime de Berkeley este disponible en todas las máquinas del cluster. Esto es útil también para otras aplicaciones, como por ejemplo mpich, y nos permite tener los programas instalados en una máquina nada más.

Primero crearemos dos carpetas en el directorio raíz. En una instalaremos las aplicaciones, la otra carpeta la usaremos para ubicar los archivos.
Utilizaremos Nfs para compartir dichas carpetas. Primero verificaremos que tenemos instalados los paquetes nfs-kernel-server y nfs-common en la máquina que va a ser el nodo maestro y las que van a ser nodos esclavos.
Después en dicha máquina abrimos Compartición del sistema / Comparticiones en el menú de KDE.

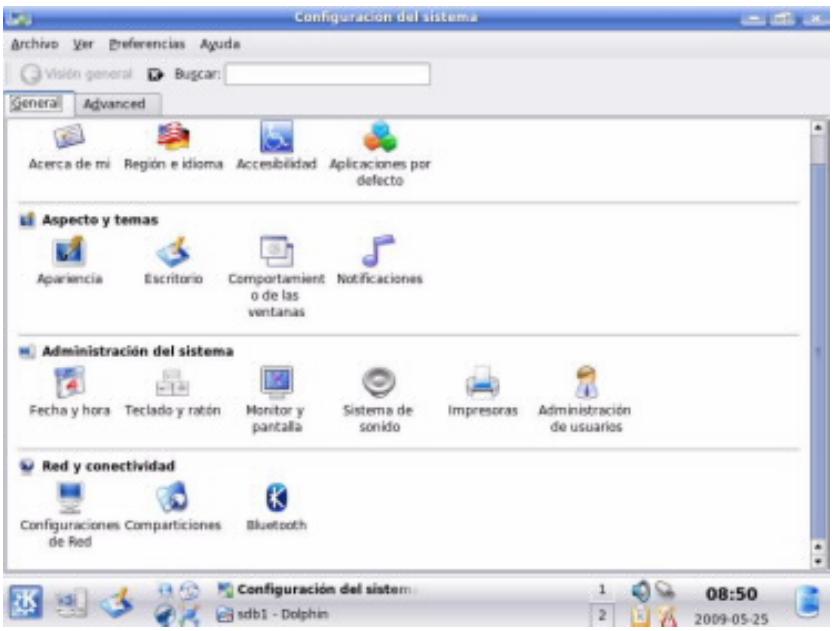

Figura 22: Panel de coonfiguración de KDE

Ejecutamos el modo administrador y seleccionamos las carpetas Aplicaciones y Archivos para compartir con Nfs.

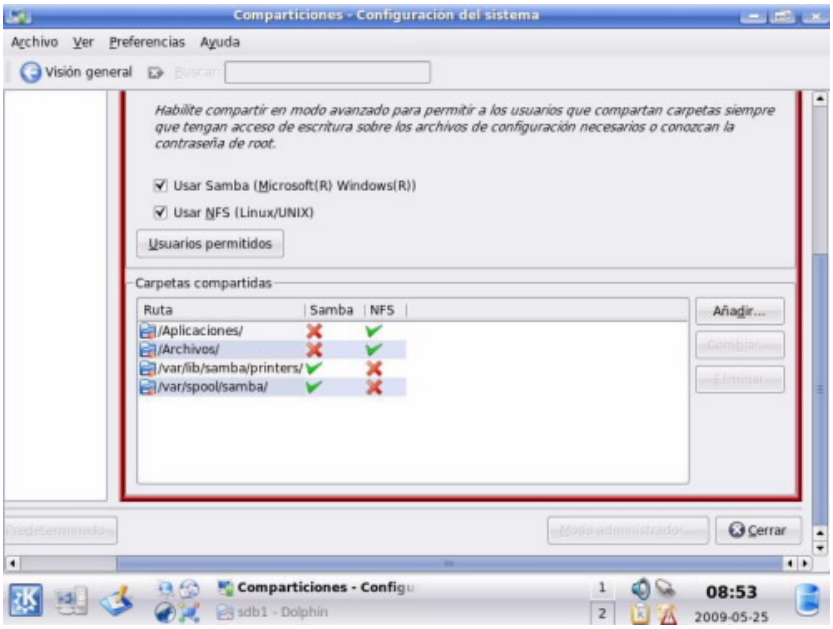

Figura 23: Panel de coonfiguración de NFS

En este instante el nodo maestro esta compartiendo las dos carpetas, pero es necesario indicar a los nodos esclavos que monten dichas carpetas para tener acceso a ellas.

Para conseguir esto editamos el archivo “fstab” en la carpeta “etc” del directorio raiz. El fichero debería tener un aspecto como este:

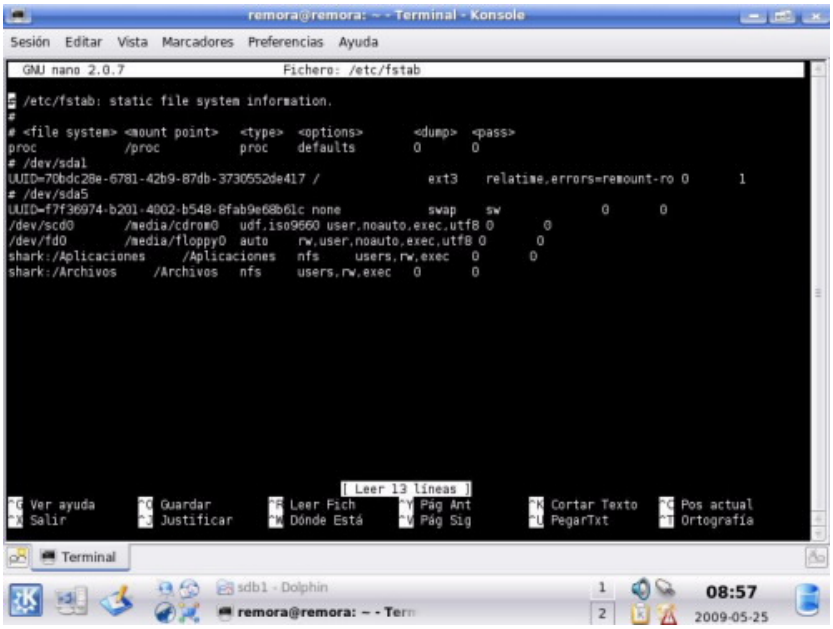

Figura 24: Edicion de fstab

Para finalizar deberemos asignar desde el nodo maestro, los permisos adecuados a las carpetas Archivos y Aplicaciones. Hay que tener en cuenta que debe existir el mismo usuario en todas las máquinas del cluster para que las aplicaciones funcionen y que debe tener permisos de lectura, escritura y ejecución.

.2.2. Instalación del Compilador GCC-UPC

Para el correcto funcionamiento del entorno de trabajo es necesario utilizar una versión del compilador de UPC compatible con el base runtime de Berkeley. Una versión correcta se puede encontrar en el siguiente enlace:

ftp://ftp.intrepid.com/pub/upc/rls/upc-4.0.3.5/upc-4.0.3.5-i686-linux-fc5.tar.gz

Descomprimimos el archivo con:

$ tar -xf <nombre_del_archivo>.tar

Editamos el archivo ".barshrc" ubicado en el directorio home del usuario. Al final del archivo añadimos la siguiente línea:

PATH=$PATH:<Ruta_al_directorio_del_UPC>/usr/local/upc/bin

Para comprobar que UPC responde bien ejecutamos el siguiente comando:

$ upc -v

Deberíamos obtener la siguiente información.

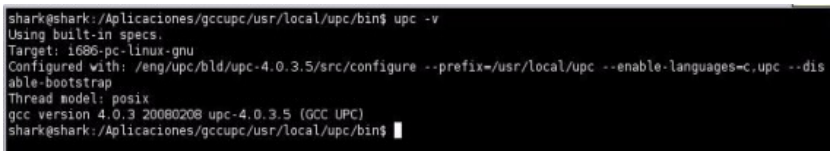

Figura 25: Resultado de lanzar el comando upc -v

.2.3. Instalación del base runtime de Berkeley

Para el correcto funcionamiento del entorno de trabajo es necesario utilizar una versión del compilador de UPC compatible con el base runtime de Berkeley. Una versión correcta se puede encontrar en el siguiente enlace:

http://upc.lbl.gov/download/release/berkeley_upc-2.8.0.tar.gz

Descomprimimos el archivo con:

$ tar -xf <nombre_del_archivo>.tar

Verificamos que tenemos instalado el compilador upc con:

$ upc -v

Deberíamos tener una salida parecida a esta:

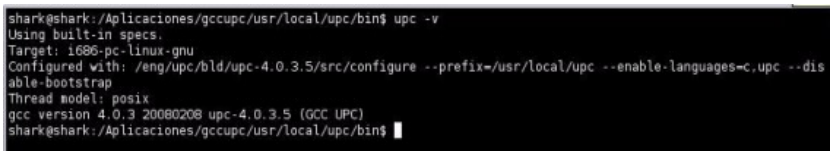

Figura 26: Resultado de lanzar el comando upc -v

Creamos una carpeta para almacenar la instalación. En ella creamos tres subcarpetas; wrk, src, rls.

En src, descomprimiremos el archivo del base runtime. En wrk realizaremos la configuración que quedará instalada en la carpeta rls.

Desde la carpeta wrk, ejecutamos el siguiente comando:

$ sudo env GCCUPC_TRANS=/Aplicaciones/gccupc/usr/local/upc/bin/upc ../src/berkeley_upc-2.8.0/configure –with-multiconf=dbg_gccupc,opt_gccupc –prefix=/Aplicaciones/bruntime/rls

Para ver que la instalación ha ido bien ejecutamos el siguiente comando:

$ upcc –version

Deberíamos obtener una salida como esta:

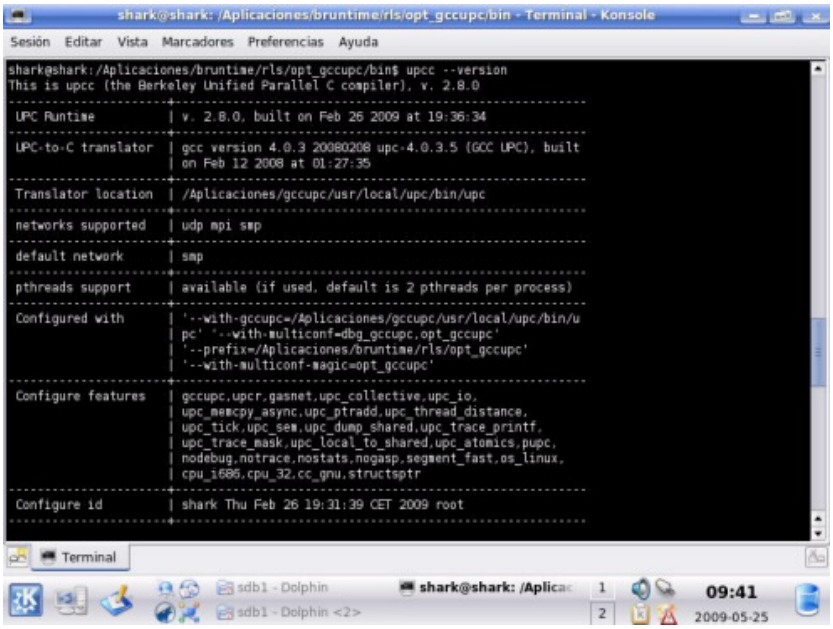

Figura 27: Resultado de lanzar el comando upcc –version

.2.4. Gestor Sun Grid Engine

Sun Grid Engine (SGE) es un sistema de colas de código abierto, desarrollado y soportado por Sun Mycrosystems. Aunque nosostros vamos a hablar de la versión de código abierto, Sun vende una versión comercial basada en SGE, la cual se la conoce como N1 Grid Engine (N1GE).

SGE se utiliza normalmente en granjas de computadores o en clusters de máquinas de alto rendimiento computacional. A su vez, SGE es responsable de aceptar, distribuir, despachar y manejar la ejecución remota y distribuida de una gran cantidad de trabajos de usuario ya sean interactivos o paralelos. También gestiona y distribuye toda una serie de recursos como pueden ser el conjunto de procesadores, la memoria, el espacio en disco, las licencias de software, etc.

Los sistemas oficialmente soportados son:

- Linux x86, kernel 2.4, glibc ¿= 2.2
- Linux AMD64 (Opteron), kernel 2.4, glibc ¿= 2.2
- Silicon Graphics IRIX 6.5
- Sun MicrosystemsSolaris (Sparc and x86) 7, 8 en 32bit y 64bit
- AppleMac OS/X, CompaqTru64 Unix 5.0, 5.1, Hewlett PackardHPUX 11.x, IBM AIX 4.3, 5.1

Con un sistema de colas como SGE, podemos dar un uso más efectivo al cluster de máquinas (repartiendo la carga por ejemplo), utilizar los recursos del cluster las veinticuatro horas al día, poder establecer unas reglas relacionadas con el número de procesadores necesarios, la memoria a utilizar, etc.

Descarga e instalación del software

La dirección donde descargar SGE es la siguiente: http://gridengine.sunsource.net

En dicha dirección podremos descargar el binario de la plataforma correspondiente, o los ficheros fuente para compilar.

Hay que definir un nodo maestro para poder instalar SGE. Una vez descargado el software el nodo maestro puede ser instalado ejecutando:

```
./install_qmaster -afs
```

Los demás nodos (nodos de ejecución y de control) deben de ejecutar:

/install_execd

El script de inicialización se instala automáticamente en /etc/INIT.d/rcsge

Para una instalación más detallada, se puede seguir la siguiente guía:

http://dlc.sun.com/pdf/820-0697/820-0697.pdf

Bibliografía

[1] Tutoriales sobre programación en PGAS. *S09: Programming with the Partitioned Global Address Space Model* http://upc.gwu.edu/tutorials/tutorials_sc2003.pdf

[2] *Página web de Co-Array Fortran* http://www.co-array.org/

[3] *Sitio web del proyecto Fortress* http://projectfortress.sun.com/

[4] *Web del proyecto Chapel* http://chapel.cs.washington.edu/

[5] *Página web del proyecto X10* http://x10-lang.org/

[6] Versión 1.2 de las espeficicaciones del lenguaje UPC. *UPC Language Specifications V1.2* http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf

[7] Manual de UPC. *THE GEORGE WASHINGTON UNIVERSITY UPC Manual* http://upc.gwu.edu/downloads/Manual-1.2.pdf

[8] *Página web de Intrepid* http://www.intrepid.com/upc.html

[9] Draft de Bonachea sobre instrucciones de copia avanzadas. *UPC Library Extensions for Explicitly Non-blocking and Non-contiguous Memcpy* http://upc.gwu.edu/ upc/upcworkshop04/bonachea-memcpy-0904-final.pdf

[10] Propuesta de primitivas de sincronización avanzadas para UPC. *Efficient Point-to-Point Synchronization in UPC* http://upc.lbl.gov/publications/PGAS06-p2p.ppt

[11] Modern Information Retrieval. *Ricardo Baeza-Yates , Berthier Ribeiro-Neto, Ed. Addison Wesley 1999*

[12] Ley Empírica de Zipf. *Zipf's Law* http://en.wikipedia.org/wiki/Zipf

[13] Prototipo original de Google. *The Anatomy of a Large-Scale Hypertextual Web Search Engine* http://infolab.stanford.edu/ backrub/google.html

[14] Google's PageRank and Beyond: The Science of Search Engine Rankings *Amy N.Langville y Carl D.Meyer, Princeton University Press,2006*

[15] Proyecto de Fin de Master de Investigación en Informática sobre Máquinas de búsqueda. *Esquema de paralelización híbrida para Máquinas de Búsqueda, Carolina Bonacic Castro, 2008*