Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

Esquema de Paralelización Híbrida para Máquinas de Búsqueda

Carolina Bonacic Castro

Director: Manuel Prieto

Proyecto de Fin de Máster en Investigación en Informática

Madrid, Julio de 2008

Resumen

Con la irrupción de las CPU multicores (*Chip-level MultiProcessor - CMPs-*) se hace imprescindible desarrollar técnicas que aprovechen las ventajas de los CMPs para aumentar el rendimiento de las aplicaciones, haciendo uso de la computación paralela.

En esta tesis se propone el diseño de una máquina de búsqueda capaz de explotar el nivel de paralelismo disponibles en los los CMPs, para el procesamiento de miles de consultas por unidad de tiempo. En particular, para esta aplicación y dada la enorme cantidad de recursos computacionales que demanda, es importante desarrollar estrategias paralelas que sean capaces de aprovechar eficientemente el hardware disponible.

El diseño propuesto utiliza técnicas de computación paralela y distribuida para organizar y procesar las consultas. Se propone un esquema de paralelización híbrida basado en los paradigmas de programación BSP y OpenMP que ha sido diseñado para sacar el máximo provecho de las características multi-threading de los CMPs para máquinas de búsqueda. Se describen implementaciones y experimentos realizados sobre dos tipos de procesadores: UltraSPARC T1 de Sun Microsystem y dos nodos Intel Quad-Xeon.

Índice general

1. Introducción **1**

1.1. Objetivos cumplidos y metodología empleada 3

1.1.1. Objetivo General 3

1.1.2. Objetivos Específicos 3

1.1.3. Metodología 3

1.2. Organización de la tesis 5

2. Trabajo Relacionado **6**

2.1. Modelo de Computación Paralela 6

2.1.1. The Bulk-Synchronous Parallel Model: BSP 6

2.2. Máquinas de Búsqueda y Clusters 8

2.3. Indices Invertidos 12

2.3.1. Distribución por documentos 13

2.3.2. Distribución por términos 13

2.3.3. Diferencias entre ambas estrategias 13

2.4. Ranking de documentos 14

2.5. Multiprocesadores 17
2.5.1. Procesador UltraSPARC T1 19
2.6. Revisión bibliográfica 21

3. Arquitectura Propuesta 24
3.1. Conceptos básicos 24
3.2. Diseño del Buscador 25
3.3. Descripción de una máquina de búsqueda 28
3.3.1. Distribución del índice invertido 28
3.3.2. Procesamiento de una consulta 29
3.3.3. Ranking iterativo 30
3.4. Esquemas de paralelización 31

4. Esquemas de paralelización: Sync/Async 33
4.1. Plataforma experimental 35
4.1.1. Ranking en punto fijo 35
4.1.2. Datos de entrada 36
4.2. Esquemas de paralelización 37
4.2.1. Modo Sincrónico 37
4.2.2. Modo Asíncrono 38
4.3. Resultados de la ejecución 39
4.3.1. Modo Síncrono 40
4.3.2. Modo Asíncrono 41
4.3.3. Aritmética de punto fijo. 42

5. Sistema Híbrido **45**

5.1. Propuesta de Paralelización 45

5.2. Resultados Experimentales 46

5.2.1. Datos de entrada 47

5.2.2. Base Experimental 47

5.2.3. Resultados 48

6. Conclusiones **52**

A. **54**

A.1. Plataforma experimental 54

A.2. Medidas de desempeño 54

A.3. Validación de las base de datos 56

Índice de figuras

2.1. Modelo BSP 8

2.2. Arquitectura de memoria distribuida. 9

2.3. Estructura de un índice invertido 12

3.1. Componentes de una máquina de búsqueda 25

3.2. Clases de la arquitectura propuesta 27

4.1. Eficiencia de un benchmark que simula el cálculo del proceso de ranking utilizando aritmética de punto fijo y flotante. . . . 36

4.2. Throughput por consulta para el modo síncrono, en dos escenarios: caso más adverso (columna gris) y el más favorable (columna negra) 40

4.3. Tiempo (sg) por consulta usando el modo asíncrono. 41

4.4. Throughput por consulta alcanzado para diferentes modos . . 42

4.5. Throughput por consulta para diferentes formas numéricas de representación: punto flotante (columna gris) y punto fijo (columna negra) 44

5.1. Speedup para dos nodos CMPs para un tráfico alto/bajo de consultas 51

A.1. Largo de la lista invertida de cada término 57

A.2. Histograma de frecuencias de los términos en el log de consultas 58

A.3. Frecuencia de cada término en todos los documentos 59

Capítulo 1

Introducción

La Web es cada vez más un inmenso repositorio de información textual [3, 9, 20]. El volumen de datos y el tráfico de consultas sobre esos datos es por lo general muy grande, incluso en buscadores especializados a un país determinado [7, 18], lo cual hace evidente la necesidad de utilizar técnicas de computación paralela [10]. La plataforma de hardware de uso común en este contexto son los clusters de PCs, lo cual conlleva a investigar el diseño de algoritmos capaces de trabajar sobre datos distribuidos con cualidades de eficiencia y escalabilidad. Esto representa nuevos desafíos en la investigación de temas tales como el diseño e implementación de máquinas de búsqueda.

En este contexto, la implementación de una máquina de búsqueda implica juntar en un mismo diseño la solución eficiente de varios problemas tales como: (1) algoritmos de distribución de tareas, los cuales son ejecutados por las máquinas broker encargadas de rutear las consultas de los usuarios a las máquinas del cluster (servidor o máquina de búsqueda), (2) algoritmos de balance dinámico de carga del trabajo realizado por los procesadores, (3) mecanismos de control de concurrencia para operaciones de lectura/escritura sobre la base de datos de texto, (4) algoritmos de compresión de datos distribuidos en varios procesadores, (5) técnicas de indexación para acelerar tiempos de respuesta a consultas, y (6) distribución de estructuras de datos

para índices y sus respectivos algoritmos de procesamiento paralelo de consultas y actualizaciones. En este último, la estructura de datos ampliamente utilizada como índice es el llamado índice invertido [8, 27, 53, 65, 66].

Actualmente la forma más difundida de realizar computación paralela es mediante el uso de bibliotecas de comunicaciones tales como PVM [29, 37, 39] o MPI [38]. Estas bibliotecas apoyan el modelo de computación de paso de mensajes el cual es utilizado en arquitecturas de memoria distribuida tales como los clusters de PCs [31]. Sin embargo, como alternativa al paso de mensaje ha surgido el modelo de computación paralela BSP (Bulk-Synchronous Parallel Model) [64] el cual tiene una estructura y metodología de diseño de algoritmos bien definida. Este modelo presenta ventajas tales como predicción del desempeño y simplicidad, y a la vez proporciona eficiencia similar a programas desarrollados bajo el esquema de paso de mensajes. Los programas pueden ser implementados utilizando bibliotecas de comunicación desarrolladas para BSP así como también bibliotecas desarrolladas para MPI o PVM. En este último caso las primitivas de comunicación MPI o PVM se utilizan para construir las primitivas de BSP.

Sobre este contexto, existe una API llamada BSPonMPI, la cual tiene las características de BSP, pero utiliza las primitivas de comunicación de MPI. En MPI aún existen muchos problemas relacionados de "deadlock", y en nuestro caso, vamos a trabajar en un escenario, donde es fundamental evitar estos problemas. En particular, este proyecto tiene como finalidad estudiar la paralelización eficiente de máquinas de búsqueda para la Web en el contexto de computación paralela sobre sistemas de alto rendimiento, como son los multicores. Más específcamente, se pueden tener dos escenarios de paralelización, que dependen del tráfico de la consultas que lleguen a la máquina de búsqueda:

- Modo Síncrono, donde la tasa de llegada es muy alta; para este caso, el modelo de programación paralela utilizado fue BSPonMPI. En este contexto, cada vez que mencionemos BSP, nos estamos refiriendo a MPI en modo síncrono.

- Modo Asíncrono, donde la tasa de llegada es muy baja. En este punto es mejor utilizar un modo de programación de paso de mensajes, MPI. El problema de "deadlook" no se presenta en este esquema.

1.1. Objetivos cumplidos y metodología empleada

1.1.1. Objetivo General

El presente trabajo de tesis tiene como objetivo el diseño, implementación y evaluación experimental de una máquina de búsqueda para la Web, aprovechando las ventajas de los multicores (Chip-level MultiProcessor -CMPs-). Esta propuesta esa basada en un esquema de paralelización híbrido con los paradigmas de programación BSPonMPI y OpenMP, que ha sido diseñado para sacar el máximo provecho de las máquinas multi-threading.

1.1.2. Objetivos Específicos

- Evaluar las ventajas y desventajas del modelo BSP para los requerimientos dados por máquinas de búsqueda en la Web.
- Proponer el diseño e implementación de una máquina de búsqueda para la Web con un alto y bajo tráfico de consultas.
- Estudiar y analizar la manera más apropiada de explotar eficientemente las ventajas de los procesadores CMPs para máquinas de búsqueda; en particular analizar el procesador Sun Microsystems' UltraSPARC T1 y el procesador multicore Intel Quad-Xeon.

1.1.3. Metodología

En general, se pretende diseñar y evaluar algoritmos implementados en BSP y OpenMP, midiendo tiempos de ejecución sobre un escenario lo más realista que sea posible.

- Estudio de la biblioteca de comunicación de BSPonMPI y completar bibliografía en el tema.
- Estudio del paradigma de programación OpenMP y completar bibliografía sobre el tema.
- Creación de base de datos de texto y log de consultas tomando una muestra de la Web Chilena desde el sitio `www.todocl.cl`.
- Implementación de algoritmos BSP y OpenMP.
- Comparación cuantitativa mediante tiempos de ejecución procesadores tipo multicores.

El desarrollo de este proyecto dio lugar a las siguientes publicaciones:

- C. Bonacic, M. Marin, C. Garcia, M. Prieto, F. Tirado "Exploiting Hybrid Parallelism in Web Search Engines", *14th European Conference on Parallel and Distributed Computing* (Europar 2008), Las Palmas de Gran Canaria, España, Aug. 26-29 2008 (Lecture Notes in Computer Science, Springer-Verlag).
- C. Bonacic, C. Garcia, M. Marin, M. Prieto, F. Tirado, "Improving Search Engines Performance on Multithreading Processors", *8th International Meeting on High Performance Computing for Computational Science* (VECPAR 2008), Toulouse, Francia, 24-27 Junio 2008.
- C. Bonacic, M. Marin, "Comparative Study of Concurrency Control on Bulk Synchronous Parallel Search Engines", *Parallel Computing 2007* (ParCo 2007), Alemania, Sept. 2007.
- M. Marin, C. Bonacic, G.V. Costa, C. Gomez, "A Search Engine Accepting On-Line Updates", *13th European Conference on Parallel and Distributed Computing* (EuroPar 2007), IRISA, Rennes, Francia, Aug. 28-31, pp. 340-349, 2007 (LNCS 4641, Springer-Verlag).

- C. Bonacic and M. Marín, “BSP Crawling sobre Clusters de Computadores”, *XVIII Jornadas de Paralelismo* (JP 2007), Zaragoza, España, 12-14 Sept. 2007.

1.2. Organización de la tesis

Este trabajo está organizado de la siguiente forma:

Capítulo 2 Se describe el modelo computacional utilizado para la paralelización de los algoritmos propuestos; se describen los tópicos básicos para el desarrollo de este proyecto de tesis. También se describe la bibliografía que sirve de base para este proyecto.

Capítulo 3 Se describe el diseño general de la máquina de búsqueda para la Web. Se definen los componentes básicos de la máquina de búsqueda; se describe un modelo orientado a objeto propuesto para la implementación describiendo su estructura de clases y los códigos C++ principales.

Capítulo 4 Se describe un modelo de paralelización, donde dependiendo del tráfico de las consultas, se aplica un modo asíncronico o síncronico sobre una máquina de búsqueda.

Capítulo 5 Se describe un modelo de paralelización híbrida que utiliza paso de mensajes para la comunicación entre los nodos y OpenMP para el proceso de ranking de las consultas.

Capítulo 6 Se presentan las conclusiones de este trabajo teniendo presente los resultados obtenidos de cada estrategia propuesta.

Apéndice A Se detallan las características técnicas de las plataformas donde fueron ejecutados los algoritmos de esta tesis; las métricas de desempeño utilizadas y la validación de las bases de datos utilizadas en cada experimento.

Capítulo 2

Trabajo Relacionado

En este capítulo se describen los tópicos considerados como base para desarrollar este trabajo de tesis.

2.1. Modelo de Computación Paralela

2.1.1. The Bulk-Synchronous Parallel Model: BSP

El modelo BSP (*The Bulk-Synchronous Parallel Model*) [62, 64] es un modelo de memoria distribuida que organiza el cómputo paralelo en una secuencia de pasos llamados supersteps. BSP puede ser visto como un modelo de programación, donde se describe el punto de vista del programador del sistema distribuido o paralelo, o como un modelo de computación utilizado en el diseño de algoritmos, el cual tiene asociado un modelo de costos que permite predecir el desempeño de los algoritmos.

BSP puede ser expresado en una gran variedad de sistemas y lenguajes de programación. Los programas BSP pueden ser escritos utilizando librerías de comunicación existentes como PVM [37, 39] ó MPI [38]. Lo único que

requiere, es que provean un mecanismo de comunicación no bloqueante y una forma de implementar la sincronización por barreras.

Un programa BSP es iniciado por el usuario en una máquina, y luego este programa se duplica automáticamente en las máquinas restantes que conforman el cluster. Cada uno ejecuta el mismo código pero con sus datos locales. Entre las librerías de comunicación especialmente escritas para el modelo BSP se encuentran: BSP lib [36], BSP pub [40] y la utilizada en este trabajo de tesis BSPonMPI [35].

La idea fundamental de BSP es la división entre computación y comunicación. Se define un step como una operación básica que se realiza sobre datos locales de un computador. Todo programa BSP consiste en un conjunto de steps que dan forman a los supersteps. Durante cada superstep los computadores trabajan sobre datos almacenados en su memoria y envían mensajes hacia otros computadores. El fin de cada superstep está dado por la sincronización de todos los computadores, punto en el cual se produce el envío efectivo de los mensajes. Los computadores continúan con el siguiente superstep una vez que todos han alcanzado el punto de sincronización y los mensajes han sido entregados en sus destinos.

La figura 2.1 muestra una máquina BSP genérica, la cual se define como:

- Un conjunto de pares procesador-memoria.
- Una red de comunicaciones que permite la entrega de mensajes punto a punto.
- Un mecanismo de sincronización de los procesadores en los supersteps.

Una máquina BSP queda caracterizada por el ancho de banda de la red de interconexión, el número de procesadores, sus velocidades y por el tiempo de sincronización de los procesadores. Todas estas características forman parte de los parámetros de una máquina BSP.

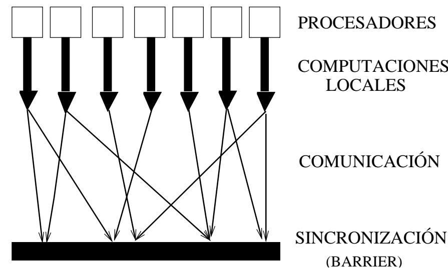


Figura 2.1: Modelo BSP

2.2. Máquinas de Búsqueda y Clusters

Lo que hacen los buscadores actuales es utilizar muchos computadores para resolver los distintos pasos involucrados en la producción de una respuesta a una consulta de usuario. A este conjunto de computadores se les llama *cluster*.

Un cluster está compuesto de un conjunto de computadores interconectados mediante una red de alta eficiencia que les permite enviarse mensajes entre ellos (ver figura 2.2).

Estos mensajes se utilizan para recolectar la información necesaria para resolver una determinada tarea como por ejemplo la solución a una consulta de un usuario. En el cluster cada computador tiene su propia memoria RAM y disco para almacenar información. Cada computador puede leer y escribir información en su propia memoria y si necesita información almacenada en otro computador debe enviarle un mensaje y esperar la respuesta.

En un cluster utilizado como máquina de búsqueda, cada computador almacena una parte de la información del sistema completo. Por ejemplo, si tenemos una colección de texto que ocupa n bytes y tenemos un cluster con P computadores, entonces podemos asignar a cada uno de los P computadores una fracción n/P de los bytes de la colección. En la práctica si la colección completa tiene n_d documentos o páginas Web, entonces a cada nodo del cluster se le asignan n_d/P documentos.

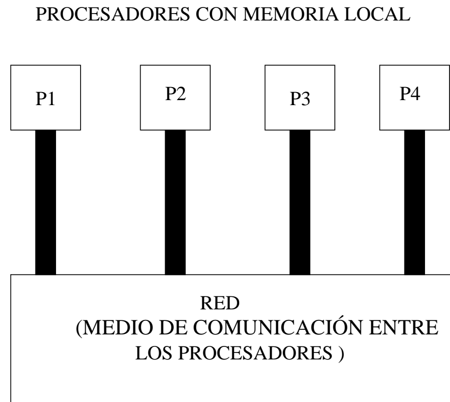


Figura 2.2: Arquitectura de memoria distribuida.

En una máquina de búsqueda las consultas de los usuarios llegan a una máquina recepcionista llamada broker, el cual distribuye las consultas entre los P nodo que forman el cluster.

Dado que cada nodo del cluster tiene un total de n_d/P documentos almacenados en su memoria, lo que hacen las máquinas de búsqueda más conocidas, es construir un índice invertido (detalles en la siguiente sección) en cada nodo con los documentos almacenados localmente en cada uno de ellos. Un índice invertido permite acelerar de manera significativa las operaciones requeridas para calcular las respuestas a las consultas de los usuarios. Cada vez que el broker recibe una consulta de un usuario, este envía una copia de la consulta a todos los nodos del cluster. El broker puede agrupar n_q consultas y tratarlas como un conjunto el cual puede ser enviado a todos los procesadores utilizando el algoritmo de broadcast $O(n_q + n_q\,G + L)$ donde G y L representan el costo de comunicación y sincronización respectivamente. En el siguiente paso, todos los nodos en paralelo leen desde su memoria las listas invertidas asociadas con las palabras [1] que forman la consulta del usuario. Luego se realiza la intersección de las listas invertidas para determinar los documentos que contienen todas las palabras de la consulta.

[1]En varios puntos del texto nos referimos a estas palabras como *términos*, en general usamos este nombre para las palabras *relevantes* en la base de texto.

Al finalizar este paso todos los nodos tienen un conjunto de respuestas para la consulta. Sin embargo, la cantidad de respuestas puede ser inmensamente grande puesto que las listas invertidas pueden llegar a contener miles de identificadores de documentos que contienen todas las palabras de la consulta. Es necesario hacer un ranking de los resultados para mostrar los mejores K resultados al usuario como solución a la consulta.

Para realizar el ranking final de documentos es necesario colocar en uno de los nodos del cluster los resultados obtenidos por todos los otros. Esto con el fin de comparar esos resultados unos con otros y determinar los mejores K. Sin embargo, enviar mensajes conteniendo una gran cantidad de resultados entre dos procesadores puede consumir mucho tiempo. Es deseable reducir la cantidad de comunicación entre nodos.

Ahora, si cada procesador ha calculado los mejores resultados para la consulta considerando los documentos (listas invertidas) que tiene almacenados en su disco, entonces no es necesario enviarlos todos al nodo encargado de realizar el ranking final. Basta con enviar a este procesador los K mejores de cada uno de los P-1 procesadores restantes. Es decir, el ranking final se puede hacer encontrando los K mejores entre los $K \times P$ resultados aportados por los P procesadores.

Pero esto se puede mejorar más aun y así reducir al máximo la cantidad de comunicación entre los procesadores [46]. Dado que los documentos están uniformemente distribuidos en los P procesadores es razonable pensar que cada nodo tendrá más o menos una fracción K/P de los mejores K resultados mostrados al usuario. Entonces, lo que se puede hacer es trabajar por ciclos repetitivos o iteraciones. En la primera iteración todos los computadores envían sus mejores K/P resultados al procesador encargado de hacer el ranking final. Este nodo hace el ranking y luego determina si necesita más resultados desde los otros nodos. Si es así entonces pide nuevamente otros K/P resultados y así hasta obtener los K mejores. En el peor caso podría ocurrir que para esa consulta en particular uno de los procesadores posea los K mejores resultados que se le van a entregar al usuario, caso en que se ne-

cesitan P iteraciones para calcular la respuesta al usuario. Pero es muy poco probable que esto ocurra para todas las consultas que se procesan en una máquina de búsqueda grande. En la práctica hemos observado [46] que se requieren una o a lo más dos iteraciones para la inmensa mayoría de las consultas, lo cual permite reducir considerablemente el costo de comunicación entre los nodos del cluster.

Las máquinas de búsqueda deben ser capaces de resolver de manera eficiente una consulta cuando el tráfico es muy elevado. La mayor parte de las consultas frecuentes, son respondidas con rapidez porque se encuentran almacenadas en la cache de las máquinas. No obstante, aquellas consultas que no se encuentran en la cache del sistema, han de ser resueltas por el conjunto de procesadores. El reto es resolver lo más rápido posible una consulta y devolver al usuario los documentos asociados más relevantes (top-K). Sin embargo, un alto tráfico de consultas y gran volumen de datos asociados con la web, supone una demanda elevada de recursos hardware (utilización de procesadores, disco y ancho de banda). Por este motivo, las máquinas de búsqueda actuales incorporan una redundancia de hardware con el fin de amortizar estos efectos.

La tendencia tecnológica actual hace pronosticar que los CMPs (*Chip-level MultiProcessor*) tendrán aún mayor importancia y el número de cores en un único chip se irá incrementando paulatinamente tal y como hacen referencia la mayoría de las previsiones de fabricantes más importantes de procesadores. En la actualidad, la mayoría de los sistemas ya incorporan esta tecnología. AMD ofrece chips de cuatro cores (*Native Quad technology*) e Intel ha comenzado a incorporar el *Intel Core*^{TM} *Extreme* en su gama servidores. Por estos motivos y por el hecho de que a fecha de hoy no conocemos que existan trabajos que evalúen las bondades de esta arquitectura en el contexto de los servidores Web, es conveniente realizar un estudio en profundidad sobre la configuración más adecuada para obtener la mejor tasa de consultas procesadas por segundo.

2.3. Indices Invertidos

Consiste en una estructura de datos formada por dos partes. La tabla del vocabulario que contiene todas las palabras (términos) relevantes encontradas en la colección de texto. Por cada palabra o término existe una lista de punteros a los documentos que contienen dichas palabras junto con información que permita realizar el ranking de las respuestas a las consultas de los usuarios tal como el número de veces que aparece la palabra en el documento. Ver figura 2.3. Dicho ranking se realiza mediante distintos métodos [8]. En este trabajo utilizamos el llamado método del vector [8, 55].

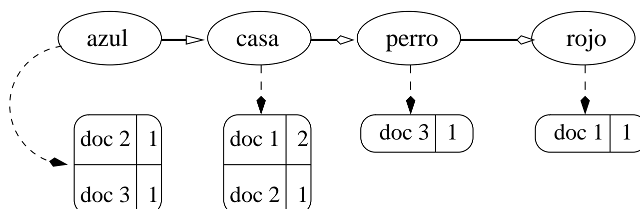


Figura 2.3: Estructura de un índice invertido

Para construir un índice invertido secuencial [28, 33, 59], es necesario procesar cada documento para extraer las palabras o términos de importancia, registrando su posición y la cantidad de veces que éste se repite. Una vez que se obtiene el término, con su información correspondiente, se almacena en el índice invertido.

El mayor problema que se presenta en la práctica, es la memoria RAM. Esta se termina antes de procesar toda la colección de texto. Cada vez que la memoria RAM se agota, se graba en disco un índice parcial, se libera la memoria y se comienza de cero con un nuevo conjunto de documentos. Al final de esta operación, se realiza un *merge* de los índices parciales, el cual no requiere demasiada memoria por ser un proceso en que se unen las dos listas invertidas para cada término y resulta relativamente rápido.

Respecto de la paralelización eficiente de índices invertidos existen varias estrategias. Las más utilizadas consisten (1) dividir la lista de documentos en P procesadores y procesar consultas de acuerdo a esa distribución y (2) en distribuir cada término con su lista completa uniformemente en cada procesador.

2.3.1. Distribución por documentos

Los documentos se distribuyen uniformemente al azar en los procesadores. El proceso para crear un índice invertido aplicando esta estrategia, consiste en extraer todos los términos de los documentos asociados a cada máquina y con ellos formar una lista invertida por procesador. Es decir, las listas invertidas se construyen basándose en los documentos que cada máquina posee. Cuando se resuelve una consulta, ésta se debe enviar a cada procesador (broadcast).

2.3.2. Distribución por términos

Consiste en distribuir uniformemente entre los procesadores, los términos del vocabulario junto con sus respectivas listas invertidas. Es decir, la colección completa de documentos es utilizada para construir un único vocabulario para luego distribuir las listas invertidas completas uniformemente al azar entre todos los procesadores. En esta estrategia, no es conveniente ordenar lexicográficamente las palabras de la tabla vocabulario, si se mantienen desordenadas, se obtiene un mejor balance de carga durante el procesamiento de las consultas. En este caso la consulta es separada en los términos que la componen, los cuales son enviados a los procesadores que los contienen.

2.3.3. Diferencias entre ambas estrategias

En el índice particionado por términos la solución de una consulta es realizada de manera secuencial por cada término de la consulta, a diferencia

de la estrategia por documentos donde el resultado de la consulta es construido en paralelo por todos los procesadores. Entonces, la estrategia por documentos soporta más paralelismo (i.e., cada consulta se resuelve en paralelo), mientras que la estrategia por términos soporta mayor concurrencia (i.e., dos o más consultas pueden ser resueltas en paralelo).

Por otra parte, la estrategia de indexación por documentos permite ir ingresando nuevos documentos de manera fácil, el documento se envía a la máquina iddoc % proc y se modifica la lista local de ese procesador. Sin embargo, para el caso de ranking mediante el método del vector (descrito en la siguiente sección), se requiere que las listas se mantengan ordenadas por frecuencia, entonces no es tan sencillo modificar la lista. No obstante, para la estrategia por términos, también es necesario hacer esa actualización cuando se modifican las listas. La gran desventaja del índice particionado por términos está en la construcción del índice debido a la fase de comunicación global que es necesario realizar para distribuir las listas invertidas. Inicialmente el índice puede ser construido en paralelo como si fuese un índice particionado por documentos para luego re-distribuir los términos con sus listas invertidas.

2.4. Ranking de documentos

El método vectorial [8] es bastante utilizado en recuperación de información para hacer ranking de documentos que satisfacen una consulta. Las consultas y documentos tienen asignado un peso para cada uno de los términos (palabras) de la base de texto (documentos). Estos pesos se usan para calcular el grado de similitud entre cada documento almacenado en el sistema y las consultas que puedan hacer los usuarios. El grado de similitud calculado, se usa para ordenar de forma decreciente los documentos que el sistema devuelve al usuario, en forma de clasificación (ranking).

Se define un vector para representar cada documento y consulta:

- El vector d_j está formado por los pesos asociados de cada una de los términos en el documento d_j.
- El vector q está compuesto por los pesos de cada una de los términos en la consulta q.

Así, ambos vectores estarán formados por tantos pesos como términos se hayan determinado en la colección, es decir, ambos vectores tendrán la misma dimensión.

El modelo vectorial evalúa el grado de similitud entre el documento d_j y la consulta q, utilizando una relación entre los vectores d_j y q. Esta relación puede ser cuantificada. Un método muy habitual es calcular el coseno del ángulo que forman ambos vectores Cuanto más parecidos sean, más cercano a 0 será el ángulo que formen y en consecuencia, el coseno de este ángulo se aproximará más a 1. Para ángulos de mayor tamaño el coseno tomará valores que irán decreciendo hasta -1, así que cuanto más cercano de 1 esté el coseno, más similitud habrá entre ambos vectores, luego más parecido será el documento d_j a la consulta q.

La frecuencia interna de un término en un documento, mide el número de ocurrencias del término sobre el total de términos del documento y sirve para determinar cuan relevante es ese término en ese documento. La frecuencia del término en el total de documentos, mide lo habitual que es ese término en la colección, así, serán poco relevantes aquellos términos que aparezcan en la mayoría de documentos de la colección. Invirtiéndola, se consigue que su valor sea directamente proporcional a la relevancia del término.

Una de las fórmulas utilizadas para el ranking vectorial es la siguiente:

- Sea $\{t_1...t_n\}$ el conjunto de términos y $\{d_1...d_n\}$ el conjunto de documentos, un documento d_i se modela como un vector:

  $$d_i \rightarrow \vec{d_i} = (w(t_1, d_i), ..., w(t_k, d_i))$$

  donde $w(t_r, d_i)$ es el peso del término t_r en el documento d_i.

- En particular una consulta puede verse como un documento (formada por esas palabras) y por lo tanto como un vector.
- La similitud entre la consulta q y el documento d está dada por:

  $$0 <= sim(d, q) = \sum_t (w_{q,t} * w_{d,t})/Wd <= 1.$$

  Se calcula la similitud entre la consulta q y el documento d como la diferencia coseno, que geométricamente corresponde al coseno del ángulo entre los dos vectores. La similitud es un valor entre 0 y 1. Notar que los documentos iguales tienen similitud 1 y los ortogonales (si no comparten términos) tienen similitud 0. Por lo tanto esta fórmula permite calcular la relevancia del documento d para la consulta q.
- El peso de un término para un documento es:

  $$0 <= w_{d,t} = f_{d,t}/max_k * idf_t <= 1.$$

  En esta fórmula se refleja el peso del término t en el documento d (es decir qué tan importante es este término para el documento).
- $f_{d,t}/max_k$ es la frecuencia normalizada. $f_{d,t}$ es la cantidad de veces que aparece el término t en el documento d. Si un término aparece muchas veces en un documento, se supone que es importante para ese documento, por lo tanto $f_{d,t}$ crece. max_k es la frecuencia del término mas repetido en el documento d o la frecuencia mas alta de cualquier término del documento d. En esta fórmula se divide por max_k para normalizar el vector y evitar favorecer a los documentos más largos.
- $idf_t = log10(N/nt)$, donde N es la cantidad de documentos de la colección, nt es el número de documentos donde aparece t. Esta fórmula refleja la importancia del término t en la colección de documentos. Le da mayor peso a los términos que aparecen en una cantidad pequeña de documentos. Si un término aparece en muchos documentos, no es útil para distinguir ningún documento de otro (idf_t decrece). Lo que se

intenta medir es cuanto ayuda ese término a distinguir ese documento de los demás. Esta función asigna pesos altos a términos que son encontrados en un número pequeño de documentos de la colección. Se supone que los términos raros tienen un alto valor de discriminación y la presencia de dicho término tanto en un documento como en una consulta, es un buen indicador de que el documento es relevante para la consulta.

- $Wd = (\sum(w_{d,t}^2))^{1/2}$. Es utilizado como factor de normalización. Es el peso del documento d en la colección de documentos. Este valor es precalculado y almacenado durante la construcción de los índices para reducir las operaciones realizadas durante el procesamiento de las consultas.

- $w_{q,t} = (f_{q,t}/max_k) * idf_t$, donde $f_{q,t}$ es la frecuencia del término t en la consulta q y max_k es la frecuencia del término mas repetido en la consulta q, o dicho de otra forma, es la frecuencia mas alta de cualquier término de q. Proporciona el peso del término t para la consulta q.

2.5. Multiprocesadores

La industria informática, ha tenido siempre un objetivo primordial (fabricantes de semiconductores, fabricantes de sistemas y usuarios): la búsqueda de la velocidad.

La velocidad va unida a las prestaciones, y por lo general, la primera ha sido la principal medida para decidirse por un sistema u otro. Sin embargo, por muy evidente que parezca, y dados los límites físicos de los semiconductores, las prestaciones pueden no estar forzosamente unidas a la velocidad. Hoy es posible construir sistemas, que aún teniendo procesadores más "lentos" que otros, ofrezcan unas prestaciones significativamente superiores. Son los sistemas multiprocesador, que como su denominación indica, incorporan varios procesadores para llevar a cabo las mismas funciones.

Evidentemente, estas mejoras en el hardware, para ser funcionales, requieren importantes desarrollos en el software, y de hecho, muchos sistemas operativos admiten extensiones multiproceso (Match, SCO, Solaris, System V, etc.), que proporcionan paralelismo "en bruto" (asignando múltiples tareas a múltiples procesadores) a nivel del sistema operativo.

Las aplicaciones escritas para facilitar el paralelismo en su ejecución, incrementan significativamente las prestaciones globales del sistema; esto es lo que se denomina "multithreading", que implica dividir una sola aplicación entre varios procesadores. Sin embargo, los desarrolladores de software y programadores de aplicaciones sólo han comenzado a explorar las vastas posibilidades de incremento de prestaciones que ofrecen los sistemas con capacidades reales de proceso en paralelo.

Un conjunto de tareas puede ser completado más rápidamente si hay varias unidades de proceso ejecutándolas en paralelo. Es necesario conocer ampliamente como están interconectados los procesadores, y la forma en que el código que se ejecuta en los mismos ha sido escrito para escribir aplicaciones y software que aproveche al máximo sus prestaciones.

Para lograrlo, es necesario modificar varias facetas del sistema operativo, la organización del código de las propias aplicaciones, así como los lenguajes de programación.

Los multiprocesadores pueden ser clasificados como máquinas con múltiples y autónomos nodos de proceso, cada uno de los cuales opera sobre su propio conjunto de datos. Todos los nodos son idénticos en funciones, por lo que cada uno puede operar en cualquier tarea o porción de la misma.

El sistema en que la memoria está conectada a los nodos de proceso establece el primer nivel de distinción entre diferentes sistemas multiprocesador:

- Multiprocesadores de memoria distribuida (distributed-memory multiprocessors). Se caracterizan porque cada procesador sólo puede acceder a su propia memoria. Se requiere la comunicación entre los nodos de proceso para coordinar las operaciones y mover los datos. Los da-

tos pueden ser intercambiados, pero no compartidos. Dado que los procesadores no comparten un espacio de direcciones común, no hay problemas asociados con tener múltiples copias de los datos, y por tanto los procesadores no tienen que competir entre ellos para obtener sus datos. Ya que cada nodo es un sistema completo, por si mismo (incluso sus propios dispositivos de entrada/salida si son necesarios), el único límite práctico para incrementar las prestaciones añadiendo nuevos nodos, esta dictado por la topología empleada para su interconexión. De hecho, el esquema de interconexión, tiene un fuerte impacto en las prestaciones de estos sistemas. Además de la complejidad de las interconexiones, una de las principales desventajas de estos sistemas, como es evidente, es la duplicación de recursos caros como memoria, dispositivos de entrada/salida, que además están desocupados en gran parte del tiempo.

- Multiprocesadores de memoria compartida (shared-memory multiprocessors). Son sistemas con múltiples procesadores que comparten un único espacio de direcciones de memoria. Cualquier procesador puede acceder a los mismos datos, al igual que puede acceder a ellos cualquier dispositivo de entrada/salida. El sistema de interconexión más empleado para estos casos, es el de bus compartido (shared-bus). Tener muchos procesadores en un único bus tiene el inconveniente de limitar las prestaciones del sistema a medida que se añaden nuevos procesadores. La razón es la saturación del bus, es decir, su sobre utilización.

2.5.1. Procesador UltraSPARC T1

El procesador UltraSPARC T1 [50], previamente conocido como Niagara, es el último desarrollo de Sun Microsystems. Este procesador se caracteriza por integrar múltiples cores en un único "die" y su capacidad de ejecución de múltiples threads simultáneamente. Está orientado a un mercado muy específico formado por aplicaciones que demanden gran capacidad de procesamiento orientado a red. Típicamente, los servidores que aprovecharán

de manera óptima las características del T1 son: servidores de base de datos, servidores web, etc. Además, ofrece otra gran ventaja que es su bajo consumo.

2.5.1.1. Características

El UltraSparc T1 es un microprocesador multicore y multithread. A continuación, se muestran sus características más importantes:

- Hasta 8 cores.
- 4 Threads por core.
- Caché L1 por core.
- Caché L2 integrada en el propio chip.
- 48 bits direcciones virtuales, 40 bits direcciones reales.
- Soporta páginas de tamaño: 8k, 64k, 4M, 256M.
- No soporta SMP.
- Predicción de saltos estática.
- Una única FPU.
- No es superescalar.
- Segmentado. Pipeline de 6 etapas.
- Consumo moderado: normal 72 Watios, máximo 79 Watios.
- Frecuencia reloj 1.0 Ghz o 1.2 Ghz.
- Arquitectura UltraSPARC 2005 de 64 bits (compatible con SPARC V9).
- Criptografía por hardware. Capaz de realizar encriptación RSA.

- TCP Offload Engine (TOE).
- Virtualización.
- Interrupciones precisas.

Niagara ha sido diseñado con una jerarquía de memoria bastante atípica. En él no se busca aumentar simplemente la capacidad de las cachés para así minimizar el número de fallos de caché; sino que se ha mejorado sustancialmente los buses de comunicación, para así permitir mayor cantidad de concurrencia posible. Para ello se ha implementado un crossbar para las comunicaciones y se ha modularizado tanto la caché L2 como el acceso a memoria principal.

La caché L1 está dividida en Caché de instrucciones y Caché de datos. Está compartida por 4 threads ya que sólo incorpora una caché por core. La principal diferencia entre la caché de instrucciones y de datos es que la primera tiene el doble de tamaño que la segunda. Esto es debido a que la caché L1 de instrucciones lee dos instrucciones sucesivas por ciclo. La caché L2 esta dividida en 4 bancos para facilitar la concurrencia en las operaciones de lectura y escritura.

2.6. Revisión bibliográfica

En [5] se presentan resultados de una tesis doctoral dedicada a estudiar la paralelización eficiente de índices invertidos utilizando PVM sobre un cluster de PCs. El problema estudiado en esa tesis tiene relación con el procesamiento paralelo de consultas de sólo lectura sobre un índice invertido distribuido y el ranking de las soluciones a dichas consultas [55, 58, 60]. Trabajos de características similares han sido presentados en [19, 41, 43, 57, 63].

Se han estudiado varias técnicas sobre la paralelización de índices invertidos [2, 26], en particular, utilizando el modelo de computación paralela

BSP [23, 45] sobre clusters de PCs [36, 40, 62, 64]. Los resultados experimentales muestran que es factible alcanzar desempeño eficiente. Algunas variantes propuestas en este trabajo, tienen relación con permitir realizar en la máquina broker acciones de balance de tareas donde la métrica que se pretende optimizar es el balance del ranking final de las consultas. El algoritmo de balance utilizado, es uno basado en una de las primeras heurísticas propuestas para resolver el problema de asignación de tareas a un conjunto de máquinas [30]. No se han estudiado otras heurísticas de balance de carga [12, 13, 14, 44].

En [22] se analizan estrategias de búsquedas mediante el estudio del diseño de servidores que trabajan sobre base de datos textuales y sus aplicaciones prácticas. Para ello se toma como punto de partida, las estrategias de indexación por documentos y por términos [24, 25]; se proponen nuevas estrategias que permiten acelerar los tiempos de respuestas de los servidores, así como reducir el espacio en memoria principal requerido para mantener dichas estructuras y conocer la unidad de trabajo para poder predecir la carga de trabajo que tendrá cada procesador cuando se realice la distribución de las consultas. Los algoritmos implementados presentan una situación intermedia entre la distribución por términos y documentos, utilizando buckets que se encuentran distribuidos entre los procesadores. Con estas técnicas se logra un alto grado de paralelismo durante las operaciones de disco, necesarias para recuperar los datos del índice invertido. A su vez, esta estrategia permite obtener una reducción del costo de hacer broadcast para las consultas y así poder alcanzar un alto grado de escalabilidad.

En relación a los CMPs, la tendencia tecnológica actual, hace pronosticar que las CPU multicores tendrán aún mayor importancia y el número de cores en un único chip se irá incrementando [1, 52]. En la actualidad la mayoría de los sistemas ya incorporan esta tecnología. AMD ofrece chips de cuatro cores (*Native Quad technology*) e Intel ha comenzado a incorporar el *Intel Core* ^{TM} *Extreme* en su gama de servidores. Por estos motivos y por el hecho de que a fecha de hoy no conocemos que existan trabajos que evalúen las bondades de esta arquitectura en el contexto de los servidores Web, es conveniente

realizar un estudio en profundidad sobre la configuración más adecuada para obtener la mejor tasa de consultas procesadas por segundo [15, 16].

Capítulo 3

Arquitectura Propuesta

3.1. Conceptos básicos

La arquitectura clásica para una máquina de búsqueda en la Web contiene los siguientes componentes básicos: (1) "crawler" el cual está encargado de recorrer la Web recuperando los documentos a indexar, (2) "indexador" encargado de organizar los documentos de una manera tal que reduzca el tiempo de respuesta para las búsquedas de información, y (3) la "máquina de búsqueda" propiamente tal la cual se construye sobre un índice invertido [8, 17]. Ver figura 3.1.

Un crawler está compuesto de un "scheduler" (planificador) que decide cuáles documentos serán recuperados, un conjunto de "robots" los cuales se conectan a los sitios Web para recuperar (bajar) los documentos, y una cola de URLs la cual es utilizada por el scheduler para priorizar los documentos a ser recuperados.

El crawler se puede ejecutar locamente en la máquina de búsqueda, recorriendo la Web mediante solicitudes a los servidores y recuperando el texto de las páginas Web que va encontrando. El indexador se ejecuta localmente y mantiene un índice sobre las páginas que recupera el crawler. La máquina

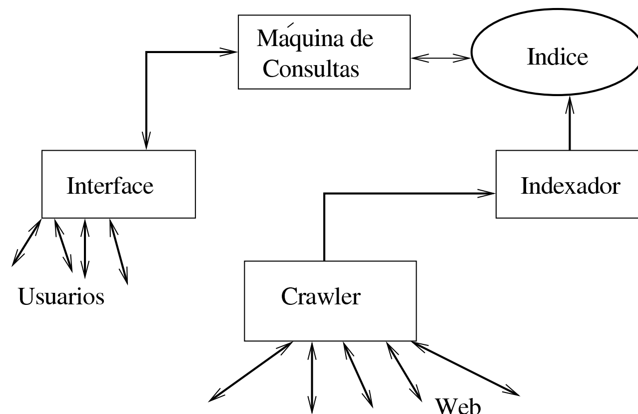


Figura 3.1: Componentes de una máquina de búsqueda

de consulta, también se ejecuta localmente y realiza la búsqueda en el índice, retornando las URLs ordenadas. La interfaz se ejecuta en el cliente (en cualquier parte de la Web) y se encarga de recibir la consulta y mostrar los resultados.

Un índice invertido consiste en una estructura de datos formada por dos partes. La tabla del vocabulario contiene todas las palabras (términos) relevantes encontradas en la colección de texto [32]. Por cada palabra o término existe una lista (invertida) de punteros a los documentos que contienen dichas palabras junto con información que permita realizar el ranking de las respuestas a las consultas de los usuarios. Dicho ranking se realiza mediante distintos métodos.

3.2. Diseño del Buscador

El diseño del buscador sigue una metodología orientada a objetos. El buscador operando en modo BSP tiene un thread principal que ejecuta un ciclo repetitivo infinito con los siguientes pasos:

```
while( true )
{ //------ Inicio de superstep

  // Recibe nuevas consultas/documentos desde el broker.
  query->rcvBroker(processor, q);

  // Recibe mensajes desde otros procesadores
  processor->recvMessagesCluster();

  // Procesa mensajes de consultas/documentos.
  processor->run();

  // Envia mensajes generados entre procesadores del cluster.
  processor->sendMessagesCluster();

  // Responde al broker las consultas finalizadas.
  processor->sendBroker();

  bsp_sync(); // Sincronizacion de procesadores.

  //------ Fin de superstep
}  //end while()
```

El buscador posee un thread encargado de gestionar la conexión TCP/IP con la máquina broker y recibir strings que representan consultas (secuencia de términos) y nuevos documentos representados como un id_doc seguido de una secuencia de términos relevantes del documento. Para gestionar los threads se utiliza la biblioteca phtreads. Los nuevos mensajes son puestos en una lista de mensajes la cual se encuentra protegida mediante semáforos (pmutex).

El thread principal se apoya en instancias de las clases Processor y Query, y a su vez la clase Processor se apoya en otras clases auxiliares. Ver diagrama 3.2. A continuación se presenta una descripción de estas clases:

Query En cada ciclo del thread principal el objeto "query" retira los nuevos mensajes desde la lista de nuevos mensajes que llegan desde la máquina broker. Se utiliza un semáforo del tipo pmutex para sincronizar los accesos concurrentes realizados por el thread encargado de la

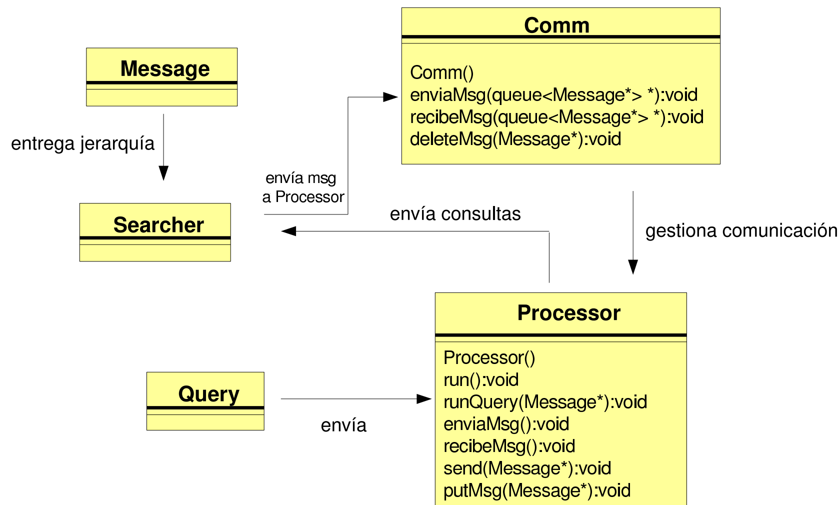


Figura 3.2: Clases de la arquitectura propuesta

comunicación con el broker y el thread principal. Las demás clases no requieren sincronización mediante semáforos.

Processor En esta clase se realizan los pasos asociados a la solución de las consultas. La clase Processor se apoya en las clases Comm, Message y Searcher. Cada objeto de clase Processor mantiene una cola de mensajes de entrada "msgin" y una cola de mensajes de salida "msgout". Durante el proceso de consultas, los nuevos mensajes destinados a otros procesadores son depositados en la cola msgout. También existe una cola similar para los mensajes de respuesta al broker ("msgoutbkr" protegida con pmutex). El método Processor::recvMenssagesCluster() almacena todos los mensajes recibidos por la clase Comm en la cola de mensajes de entrada msgin. Además, el objeto de clase Query inserta los mensajes nuevos llegados desde el broker a la cola msgin.

Comm La cual gestiona la comunicación entre procesadores. Mantiene una estructura de buffers destinada a empaquetar en un sólo gran mensaje todos los mensajes enviados desde un procesador a otro. Los mensajes son copiados byte a byte a los buffers y posteriormente se utilizan las

primitivas de la biblioteca de comunicación para realizar el envío de los mensajes grandes a sus procesadores de destino.

Message La cual es una jerarquía de clases orientada a mantener la información que se envía entre procesadores. Dependiendo de la fase en que se encuentra el proceso de una consulta dada, es necesario enviar distintos tipos de información. La clase Comm determina, según el tipo de clase en la jerarquía, la cantidad de información (bytes) a copiar desde el espacio de direcciones del objeto de la clase Message y el buffer destinado a mantener los mensajes.

Searcher Destinada a realizar el procesamiento de las consultas de los usuarios lo cual implica hacer un parsing de la consulta para extraer los identificadores de los términos, recuperar las listas de posteo desde los distintos procesadores, realizar el ranking de documentos y hacer el merge de los resultados, para luego retornar la respuesta a la máquina broker. Durante este proceso se generan uno o mas mensajes destinados a otros procesadores. Dichos mensajes quedan en los buffers de la clase Comm para su posterior envío al final del superstep.

3.3. Descripción de una máquina de búsqueda

3.3.1. Distribución del índice invertido

Las máquinas de búsqueda para la Web, utilizan un índice invertido como estructura de datos, para indexar una colección de texto y así acelerar el procesamiento de las consultas. En varios artículos se han publicado experimentos y propuestas para un procesamiento eficiente de consultas con índices invertidos [4, 6, 11, 47, 48, 49, 51, 65]. Es evidente que la eficiencia en cluster de computadores, solo se logra mediante el uso de estrategias que reducen la comunicación entre los procesadores y mantienen un balance entre la computación y la comunicación, en el proceso de resolver una consulta.

Un índice invertido, esta formado por una tabla de vocabulario y un conjunto de listas. El vocabulario, contiene un conjunto de términos que están en la colección de texto. Cada uno de los términos se asocia a una lista que contiene el identificador del documento, cuando el término aparece en la colección, junto con datos adicionales utilizados para fines del procesamiento. Para resolver una consulta, es necesario obtener el conjunto de documentos asociados a los términos de la consulta y, a continuación, realizar una clasificación de estos documentos seleccionando los K mejores como respuesta a la consulta

Las actuales máquinas de búsqueda, utilizan una partición por documentos para distribuir el índice invertido en un conjunto de P procesadores. En este caso, la colección de documentos esta uniformemente distribuida en los procesadores y el índice invertido se construye a partir de ellos en cada procesador. Resolver una consulta implica (a) tener una copia del mismo en cada procesador, (b) cada procesador extrae de manera local, los mejores K documentos asociados a los términos de la consulta y (c) se realiza un merge de los resultados de todos los procesadores, para calcular las mejores respuesta de una consulta.

Las consultas que llegan a una máquina de búsqueda, suelen ser de sólo lecturas. Esto significa que no ocurrirá que múltiples usuarios escriban al mismo tiempo información sobre el texto de la colección. Por lo tanto, no habrá problemas de concurrencia sobre los datos. La inserción de nuevos documentos se efectúa de manera off-line.

3.3.2. Procesamiento de una consulta

En la parte paralela del servidor, las consultas llegan desde una máquina recepcionista que llamamos “broker”. La máquina broker, se encarga de enviar las consultas a los procesadores del cluster (para este trabajo, cada procesador es un CMPs del cluster) y recepcionar las respectivas respuestas. El broker distribuye las consultas, de acuerdo a heurísticas de balance de

carga. Cada heurística depende del método de partición del índice invertido. En general, el broker tiende a distribuir de manera uniforme las consultas sobre los procesadores.

Más en detalle, el procesamiento de una consulta se compone básicamente de una fase en la que es necesario buscar todas las listas de documentos asociadas a un término de la consulta, y realizar una clasificación (ranking) de estos documentos, con el fin de obtener una respuesta. Nos centraremos en situaciones en la que es relevante optimizar el rendimiento de las consultas.

Un punto importante, es la forma de organizar el procesamiento de una consulta en cada procesador. Básicamente, aplicamos la combinación de dos estrategias que hemos diseñado para manejar de manera más eficiente los recursos de hardware, frente a variaciones del tráfico de las consultas:

- **Distribución tipo Round Robin** permitimos que la consulta use un "quantum" de computación, comunicación y acceso a disco antes de entregar los recursos a otra consulta.
- **Modo operacional** dinámicamente intercambiamos el modo operacional de una máquina de búsqueda entre paso de mensajes asincrónico y sincrónico, dependiendo del tráfico de consultas observado.

3.3.3. Ranking iterativo

El "ranker" es el procesador que recibe el conjunto de consultas desde broker. Todos los procesadores actúan como ranker para un subconjunto de consultas y en cada uno de ellos se extraen los documentos (fecthing) de las listas invertidas y el proceso de ranking. Cada consulta es procesada iterativamente siguiendo los siguientes dos pasos:

- **Fetching:** Este primer paso consiste en extraer los K mejores elementos de todas las listas de los términos de una consulta y enviar esta información al ranker. Es decir, el ranker envía una copia de cada

consulta a todos los P procesadores. A continuación, todos los procesadores envían los K/P pares (doc_id, frecuencia) obtenidos del ranker.

- **Ranking:** Se realiza la intersección final de los documentos proporcionados del paso anterior. Utilizamos un modelo vectorial para la ejecución del ranking, con las técnicas de filtrado propuestas en [56]. Si fuese necesario obtener más información para formar la respuesta final de una consulta, el ranker solicita una nueva iteración hasta obtener los mejores K elementos y finalizar así el proceso.

Para realizar con éxito el proceso de ranking puede ser necesario una o más iteraciones. En cada iteración, los nuevos pares (doc_id, frecuencia) por cada término que conforman la consulta, forman la lista invertida que es enviada al ranker. El concepto de iteración es esencial para distribuir equitativamente el acceso a los recursos del sistema (tipo Round Robin): se procesan trozos de consultas de tamaño K y los documentos son organizados en iteraciones.

3.4. Esquemas de paralelización

Con el fin de hacer un buen uso de la tecnología actual, organizamos el procesamiento de las consultas en supersteps que son ejecutados por todos los procesadores en paralelo. Asumimos que existe un tráfico de consultas que llegan al broker con t términos por consulta. El algoritmo propuesto puede ser resumido de la siguiente manera:

SuperStep i (Broadcast) Cada procesador P recibe Q-m consultas del broker donde m son las consultas pendientes que necesitan una nueva iteración. Los t términos de una consulta son enviados a todos los nodos.

SuperStep $i+1$ (Fetching) Cada procesador P obtiene t·Q listas invertidas de tamaño K/P cada una, y las envía al procesador ranker.

SuperStep $i+2$ (Ranking)

Parte 1 Cada procesador P recibe t·Q·P listas de tamaño K/P y realiza una intersección de esta información por términos y consulta. Esta información es almacenada en estructuras de datos contiguas en memoria que faciliten su procesamiento.

Parte 2 Cada procesador utiliza T≤Q threads en la fase del ranking sobre un grupo de Q consultas.

Parte 3 Se actualiza el estado de la consulta, es decir, se solicita una nueva iteración si es necesaria más información de lista ó se envían los mejores R (R=K/2) resultados al broker.

Decir que es importante la organización de las consultas dentro de la máquina de búsqueda, se debe a las características actuales de los procesadores. Es un hecho evidente de los multicores presentan ventajas apropiadas para explotar el paralelismo de las aplicaciones. Por este motivo, esta tesis se basa en dos estudios principalmente.

En el primero de ellos, estudiamos la factibilidad de incorporar threads OpenMP al procesamiento de consultas; donde pueden llegar un número de consultas mayor a la cantidad de threads disponibles, o el caso contrario, donde el conjunto de consultas es menor que el número de threads. Este estudio fue realizado en un procesador UltraSPARC T1 y el análisis hecho se encuentra en el capítulo 4.

Dados los resultados obtenidos en el primer análisis, nos dimos cuenta que es factible hacer una combinación de esquemas de programación paralela para resolver de manera eficiente, una consulta en una máquina de búsqueda explotando al máximo el paralelismo que ofrecen los multicores. Entonces, en el capítulo 5 presentamos un modelo híbrido de programación BSP y OpenMP capaz de procesar un tráfico alto de consultas. En este caso, trabajamos con dos nodos multicores Intel Quad Xeon y vimos el comportamiento dentro de un nodo, y la comunicación entre ambos.

Capítulo 4

Esquemas de paralelización: Sync/Async

El diseño y ejecución de los actuales motores de búsqueda para la Web, se basa en un esquema de mensaje asíncrono, es decir, llega una consulta la cual es atendida por un thread en un esquema clásico maestro/esclavo.

Por otro lado, la cantidad de trabajo necesario para la resolución de una consulta, sigue la llamada ley de Zipf, que en la práctica significa que algunas consultas, en particular, las compuestas por los términos más populares, van a requerir más tiempo de procesamiento, mientras que las que contengan términos menos frecuentes, demandarán un tiempo mucho menor de procesamiento.

Dado este enfoque, podemos determinar que resolver un conjunto muy grande consultas, va a consumir un número considerable de recursos por procesador, como son el disco, ciclos del procesador, ancho de banda, entre otros.

Sin embargo, hemos encontrado un diseño donde las principales etapas para procesar una consulta, son distribuidas de una manera en que éstas comparten los recursos del cluster uniformemente aplicando un esquema

Round Robin[48, 49]. Este esquema puede ser realmente útil cuando el tráfico de consultas es muy variable, dentro de una máquina de búsqueda.

En particular, hemos observado que para un esquema estándar asíncrono de procesamiento de consultas, cuando el tráfico de las mismas es muy elevado, puede ser perjudicial para el rendimiento en general, debido a la distribución de Zips de las cargas de trabajo. También hemos observado, que el método Round Robin de procesamiento de consultas, resuelve este problema de manera eficiente. Hemos validado esta hipótesis a través de experimentos realizados sobre un log real de consultas de un 1 TB de la Web.

En este capítulo, hemos hecho un estudio de diferentes estándares, implementando una máquina de búsqueda, basados en el modelo de distribución de consultas Round Robin, sobre un sistema ChipMultithreading [54] y OpenMP.

Como plataforma experimental, hemos optado por un procesador UltraSPARC T1 de Sun Microsystem (Niagara)[42]. Este procesador, simboliza el reciente cambio de los CMPs en los servidores y presenta un enfoque radical que permite aprovechar el rendimiento de computación y escalabilidad con un bajo consumo de electricidad.

Para fines de programación, T1 puede ser visto como un conjunto de procesadores lógicos que comparten algunos recursos. Se podrá pensar que otros esquema de paralelización de memoria compartida, tales como los sistemas SMP, que también son buenos candidatos para este proceso. Sin embargo, la distribución de los recursos introducidos en el T1 para aumentar la utilización de éstos, puede causar serios cuellos de botella, y por lo tanto, estrategias que son apropiadas para estas máquinas pueden ser inadecuadas o menos eficientes. Uno de los objetivos que motivan este estudio es analizar una aplicación paralela para máquinas de búsqueda, bajo este punto de vista.

4.1. Plataforma experimental

Como plataforma de experimentación, hemos optado por un procesador UltraSPARC T1 de Sun Microsystems, cuyas caracteríscas más importantes se encuentran en el capítulo 2 y apéndice A.

En esencia, el T1 es un procesador multithreading de grano fino (FGM) [34] que cambia los threads de ejecución de cada ciclo para ocultar las ineficiencias provocadas por largas latencias, como accesos a memoria [61]. Un único thread de aplicación, funciona mejor en procesadores tradicionales, pero los multithreading puede beneficiarse de esta arquitectura: cada thread es más lento, pero esta arquitectura mejora el rendimiento del uso de recursos.

En nuestra implementación, el nivel de paralelismo, ha sido explotado por medio de OpenMP, que es soportado por los compiladores nativos de Sun.

4.1.1. Ranking en punto fijo

El procesador UltraSPARC T1, solo proporciona una unidad de punto flotante compartida por sus 8 cores en el chip, es decir, un solo thread puede utilizarlo a la vez. Por otra parte, aunque sólo un thread utiliza la unidad de punto flotante, existe una penalización de 40 ciclos de acceso a la unidad. La mayoría de las aplicaciones comerciales tienen poco contenido de punto flotante, por lo tanto no es una desventaja. Sin embargo, en nuestra aplicación, una de las fases más costosas del proceso del ranker, utiliza aritmética de punto flotante para clasificar los documentos más relevantes para una consulta.

Por tal motivo, hemos modificado el ranker, para evitar aritmética de punto flotante. Nuestra aplicación utiliza 32 bits de punto fijo de representación de datos para las listas invertidas, y realiza cálculos usando una configuración de la aritmética de punto fijo. El introducir punto fijo en nues-

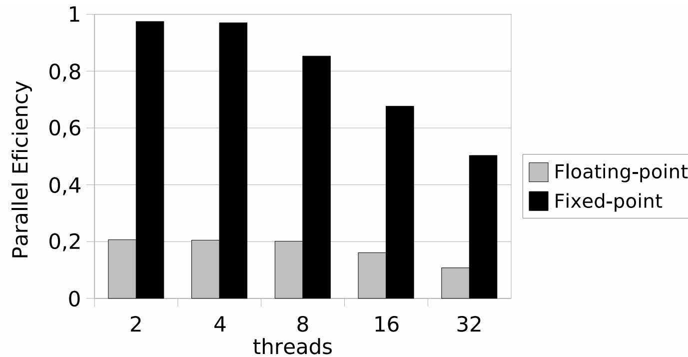


Figura 4.1: Eficiencia de un benchmark que simula el cálculo del proceso de ranking utilizando aritmética de punto fijo y flotante.

tra aplicación, produce un overflow de un 20 a 40 %, pero el generado por el punto flotante (la instrucción "sqrt" tarda mil ciclos) compensa este costo.

La figura 4.1 ilustra los beneficios potencias de esta optimización. Se muestra la escalabilidad de un benchmark sintético que intenta simular el tipo de cálculo realizado por el proceso de ranking, que mezcla diferentes operaciones aritméticas como divisiones, multiplicaciones, logaritmos y raíces cuadradas. Como era de esperar, los resultados de la versión de punto flotante no son buenos, en cambio la versión de punto flotante, escala razonablemente.

4.1.2. Datos de entrada

Todos los resultados de este trabajo se han obtenido utilizando una base de datos de la web chilena, muestra tomada de `www.todocl.cl`. La colección de texto, contiene alrededor de un millón de términos en español (1,5 GBytes de tamaño). Las consultas han sido seleccionadas al azar del log, que contiene 127,000 consultas. Más información, ver apéndice A.

4.2. Esquemas de paralelización

Como se mencionó anteriormente, las arquitecturas de chip multithreading introducen un nuevo escenario a nivel de paralelismo, donde los threads se convierten en la clave para lograr un buen rendimiento. En este sentido, algo fundamental es el modo de funcionamiento de una máquina de búsqueda:

- **Modo asincrónico:** para un bajo tráfico, cada consulta es atendida por un único thread (master). Dicho thread puede comunicarse con otros P thread (slave), localizados en los P nodos del cluster.
- **Modo sincrónico:** para un alto tráfico, todos los threads activos son bloqueados y solo un thread tiene el control del proceso de un conjunto de consultas. En este caso, los mensajes son insertados en el buffer de todos los nodos del cluster, que son enviados al comienzo de cada iteración, punto en el cual todos los procesadores son sincronizados. En esta estrategia, los recursos del sistema son utilizados de mejor manera por la forma de distribuir los threads y el costo de sincronización que se reduce significativamente y la comunicación, que es ejecutada en bloques.

A continuación, describimos estos dos modos, con mayor precisión.

4.2.1. Modo Sincrónico

El paralelismo de grano grueso, presentado para el modo síncrono puede ser fácilmente expresado por medio de directivas de OpenMP y utilizando métodos convencionales de distribución de consultas. Sin embargo, ejecutar el mismo código con diferentes threads para resolver una sola consulta, puede causar problemas de los recursos compartidos de procesador T1, sobre todo el espacio de caché y el ancho de banda de la memoria. Hemos tratado de minimizar esto, distribuyendo consultas con términos similares en el mismo

procesador. En esencia, nuestra idea es aprovechar la localidad, a fin de aumentar la cooperación entre los threads y evitar los costosos accesos a memoria, tanto como sea posible.

El algoritmo 1 muestra el pseudo-código de nuestra propuesta. Como primer paso, la máquina broker (master), intenta reunir las consultas con términos comunes y a continuación, cada procesador (slave) encuentra los mejores documentos asociados.

Algorithm 1 Máquina de búsqueda operando en modo síncrono

```
En todas las máquinas broker:
// grupo de consultas con términos similares
query_batch = build_clustered_batch(current_queries);
broadcast_to_all_processors(query_batch);

En cada procesador:

#pragma omp parallel for private(...) shared(...)
for q=1:Nqueries_processor do

  query = query_batch[q];
  for term=1:terms_in_query do
    posting_list[term] = fetch(term);
  end for
  best_docs[query] = ranking(posting_list, terms_in_query);

end for
...
```

4.2.2. Modo Asíncrono

Para el modo asíncronico, utilizamos los threads de OpenMP para realizar el ranking de los documentos y así obtener los resultados de las consultas. Este tipo de paralelismo, implica ejecutar threads OpenMP en diveros puntos de la rutina. En particular, para los casos de operaciones idénticas, donde debe ejecutarse la lista completa de cada término de la consulta, en forma

paralela cada threads trabaja sobre una parte de la lista. La técnica de filtrado, es un poco más complicada. Se necesita una sincronización por barrera para actualizar ciertos parámetros. Los nuevos documentos extraídos deben esperar esta sincronización para ser incluidos dentro la respuesta de la consulta. La sincronización debe ser actualizada de manera simultánea por los threads. Entonces, a través de secciones críticas se trata de resolver este problema de actualización; aunque esto ocurre con menor frecuencia durante el procesamiento de las listas.

Algorithm 2 Máquina de búsqueda operando en modo asíncrono

```
En todas las máquinas broker:
// Envío de consultas
dispatch_to_processor_P(query);

En cada procesador:

for term=1:terms_in_query do
   #pragma omp parallel private(...) shared(...)
   posting_list[term] = parallel_fetch_and_operations(term);
end for
best_docs = parallel_ranking(posting_list, terms_in_query);
...
```

4.3. Resultados de la ejecución

En esta sección, se intenta dar respuesta a las preguntas antes planteadas. El estudio que hemos hecho, puede ser considerado como una primera aproximación de un sistemas más complejo, basado en procesadores CMPs, que deben comportarse como esclavos en nuestro contexto. Nuestro objetivo es analizar en detalle los enfoques paralelos que hemos planteado y esbozar algunas conclusiones preliminares que pueden ser extrapoladas a una infraestructura más real.

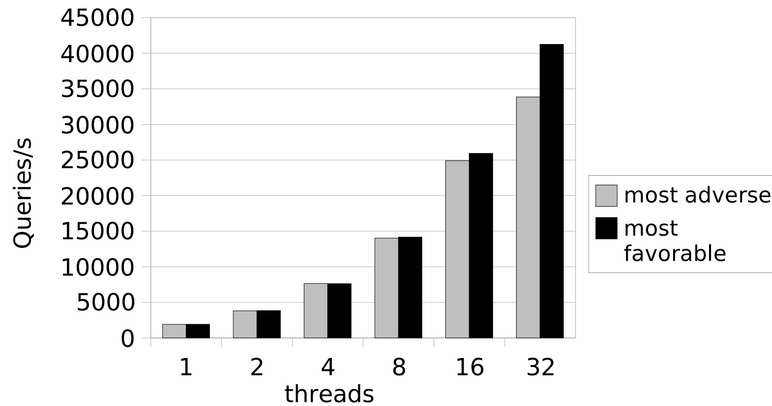


Figura 4.2: Throughput por consulta para el modo síncrono, en dos escenarios: caso más adverso (columna gris) y el más favorable (columna negra)

4.3.1. Modo Síncrono

La figura 4.2 muestra el rendimiento alcanzado por nuestra propuesta sincrónica. Como fue mencionado anteriormente, hemos tratado de mejorar la cooperación entre threads con datos distribuidos y threads que atienden a grupos de consultas con términos similares. La columna gris corresponde a la situación más adversa: no hay términos comunes entre las consultas y todos los threads compiten por los recursos disponibles. La columna en negro, corresponde a la situación más favorable: todos los threads atienden a un conjunto de consultas con términos idénticos.

Al ejecutar 16 y 32 threads, vemos las diferencias entre ambos escenarios, donde se destacan los beneficios de trabajar con threads distribuidos. En cualquier caso, el rendimiento es bastante satisfactorio en ambos escenarios. El speedup aumenta proporcionalmente con el número de threads, y en el escenario más favorable, nuestra máquina de búsqueda síncrona alcanza un speedup de 22 utilizando 32 threads.

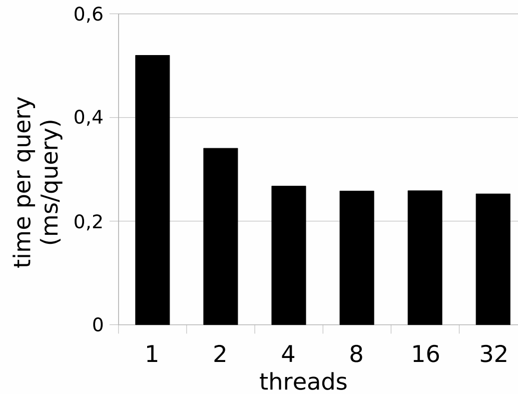


Figura 4.3: Tiempo (sg) por consulta usando el modo asíncrono.

4.3.2. Modo Asíncrono

Los resultados experimentales (véase la figura 4.3) muestran que la ganancia proveniente del paralelismo, no es realmente significativa. Esto se debe principalmente, al hecho que en cada iteración tipo Round Robin de una determinada consulta, la cantidad de datos (los que participan en las listas invertidas de tamaño K) que se procesa, es pequeña. Por ejemplo, en el proceso de ranking, es necesario ordenar los documentos; entonces, al tratar de hacer una clasificación en paralelo utilizando OpenMP, no es significativo, ya que la cantidad de documentos rankeados, no son lo suficientemente grandes.

Recordemos que utilizar Round Robin es necesario para evitar que las consultas muy largas consuman todos los recursos, en desmedro de las consultas pequeñas. Además, la técnica de filtrado, se utiliza para evitar trabajar con la lista completa de cada término; se debe actualizar una barrera de sincronización para obtener el ranking final y decidir que documentos son los más relevantes. Este barrera de sincronización, pasa a ser una sección crítica para los threads OpenMP, por lo tanto esta serialización, introduce una degradación en el rendimiento.

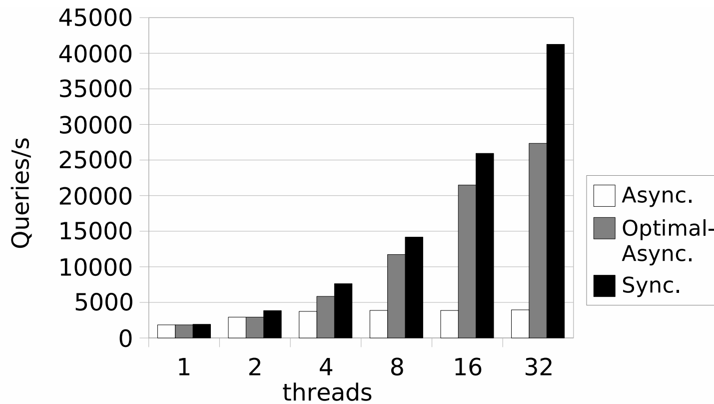


Figura 4.4: Throughput por consulta alcanzado para diferentes modos

Las actuales máquinas de búsqueda son completamente asíncronas y propensas a estos problemas. En general, estas máquinas utilizan técnicas de rastreo para evitar recuperar la lista completa de documentos y, por lo tanto, son escencialmente las mismas dificultades para obtener las ventajas que proporcionan los sistemas CMPs. En este sentido, puede afirmarse que para un alto tráfico de consultas, la ejecución de múltiples consultas también se superpone al modo asíncrono y esto proporcionaría suficiente paralelismo. A este caso, le hemos llamado *Óptimo-Async (OA)*.

La figura 4.4 destaca que incluso en este caso, el modelo asíncrono se comporta peor que el modo síncrono. La figura muestra el throughput logrado en los diferente modos. A pesar que el modo OA explota ambas maneras de paralelismo - paralelismo dentro y fuera de la consulta - en un escenario óptimo. Por otra parte, para un determinado número de threads, el rendimiento logrado por el modo OA no supera al modelo sincrónico.

4.3.3. Aritmética de punto fijo.

La figura 4.5 analiza la escalabilidad de la aritmética de punto fijo. Se muestra el número de consultas por segundo que se pueden resolver sobre un

escenario síncrono cuando se realiza el ranking de los documentos. Como era de esperar, la versión para punto flotante no escala más allá de un número pequeño de threads.

Como último punto, es necesario analizar si las operaciones de punto fijo produce algún efecto en (1) el conjunto de documentos seleccionados como la respuesta a una consulta y (2) su posición relativa dentro de los mejores K resultados. Se evaluó experimentalmente ejecutando el proceso de ranking, para punto fijo y flotante de un conjunto de consultas.

Hemos realizado dos pruebas para el ranking de documentos generados en ambos casos; para el caso de punto fijo, lo hemos llamado A y para punto flotante B. La primera prueba calcula el ratio de $|A \cap B|/|B|$ para los cuales se obtuvieron valores muy cercanos a 1; observamos que los valores medios oscilan entre 0,99 y 1,0 para un conjunto grande consultas. Esto indica que ambos conjuntos son prácticamente idénticos.

La segunda prueba, calcula la correlación de Pearson de los conjuntos A y B para medir la posición relativa de los documentos, es decir, analizar si los mejores K documentos de A se representan en la misma posición que el conjunto B. Una vez más se obtuvieron valores muy cercanos a 1 lo que indica que casi no hay diferencia en la posición relativa de los documentos recuperados en ambos conjuntos.

Es factible explotar el nivel paralelismo de los multicores, para optimizar el trabajo de una máquina de búsqueda, independiente del tráfico de consultas que llegue. Por lo tanto, es interesante estudiar un sistema híbrido para este tipo de máquinas, donde se combine la comunicación en nodos a través de BSP y el procesamiento de las consultas, explotando los threads OpenMP, para un alto tráfico de consultas. Este estudio se ve reflejado, en el siguiente capítulo.

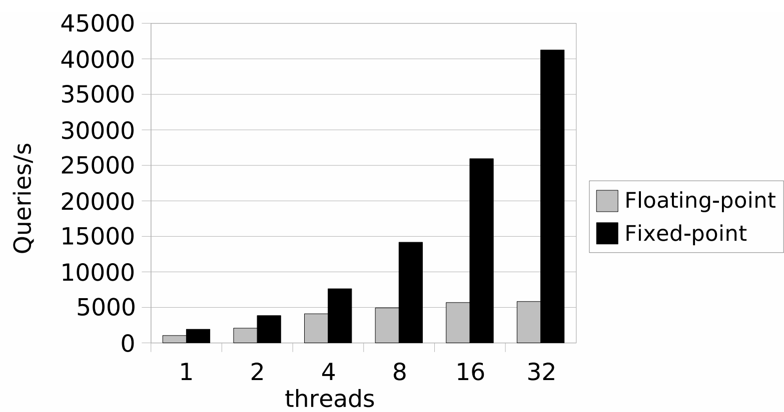


Figura 4.5: Throughput por consulta para diferentes formas numéricas de representación: punto flotante (columna gris) y punto fijo (columna negra)

Capítulo 5

Sistema Híbrido

En este capítulo proponemos un esquema de paralelización basado en BSP y OpenMP. El proceso del ranking de documentos se realiza en paralelo utilizando los threads de OpenMP. Esta fase es la parte más costosa del proceso de búsqueda y por lo tanto conviene reducir el tiempo de ejecución en escenarios con un alto tráfico de datos. Nuestro objetivo es tener T threads que trabajen en la fase de ranking con un grupo de Q consultas en un mismo nodo por un periodo de tiempo $Q \geq T$ (este proceso se realiza en paralelo a través de los P nodos disponibles en el sistema).

5.1. Propuesta de Paralelización

Como código de referencia, que hemos utilizado es el descrito en el capítulo 3; dicho código ha demostrado que consigue escalar adecuadamente en un cluster convencional con diferentes arquitecturas [47, 48, 49]. Se podría pensar que la mejor y más inmediata manera de explotar este nuevo nivel de paralelismo, es extender el número de procesos BSP a los cores disponibles. Sin embargo, anticipamos que ello conlleva la partición adicional del índice invertido a través de los cores, aumentando así el nivel de comunicación entre los nodos y el conflicto por el hardware compartido dentro del chip.

Con la aparición de los procesadores CMPs, es imprescindible realizar un estudio y analizar cual es la manera más apropiada de explotar eficientemente esta tecnología. A priori, la estrategia más sencilla sería extender el esquema ranking-fetching [46] entre los cores disponibles en un CMPs. Sin embargo, podemos anticipar que este enfoque no es el más adecuado, porque conlleva una sobrecarga importante debido a la necesidad de duplicar el índice invertido entre los BSP-threads y a la competencia de los recursos compartidos en un CMPs como la memoria caché, acceso a la memoria principal, interfaces de comunicación, etc.

Nuestra propuesta trata de mitigar estos efectos. Se basa en la idea de que el broker continúa su tarea de distribución de consultas a través de los nodos del cluster y obtiene las respectivas respuestas [48]. Cada nodo debe resolver un conjunto de Q consultas (grupo de consultas) y realizar el proceso de ranking en paralelo con OpenMP distribuyendo las consultas entre los cores. Es importante destacar que la etapa del ranking es la más costosa en términos de computación y por ello es la principal candidata a ser paralelizada.

La idea que subyace detrás de la distribución tipo Round Robin, es la de dividir el proceso de resolución de una consulta (fecthing+ranking) en fases más pequeñas para que puedan ser distribuidas fácilmente.

La parte más costosa del proceso de resolver una consulta (descrito en la sección 3.4) rea ce en el ranking, lo que justifica por si solo el uso de threads OpenMP.

5.2. Resultados Experimentales

La explotación de los threads a nivel de paralelismo se ha realizado a través de directivas de OpenMP que soportan el compilador nativo de Intel C/C++ [21]. También, es importante mencionar que hemos introducido

dentro del mismo programa, las directivas de comunicación de la libaría BSPonMPI.

Las características de la plataforma experimental utilizada en este capítulo, se encuentra detallada en el apéndice A

5.2.1. Datos de entrada

Los resultados se han obtenido utilizando una muestra de la base de datos de la web chilena tomada de `www.todocl.cl`. Las consultas han sido seleccionados al azar de un conjunto de 127,000 consultas extraídas del log (`todocl`). Más información, ver apéndice A.

5.2.2. Base Experimental

Hemos planteado dos escenarios experimentales, con un alto y bajo tráfico de consultas que llegan a los nodos del cluster. El tráfico esta determinado por el broker y es independiente de la configuración de BSP y los threads OpenMP utilizandos en la máquina de búsqueda.

Supongamos que el broker envía en promedio B consultas por unidad de tiempo. La máquina de búsqueda debe resolver estas consultas usando conjuntos de tamaño $B = Q{\cdot}P$, donde Q son las consultas por procesador que se resuelven utilizando T threads OpenMP. P representa el número de threads BSP en cada supersteps.

Asumiendo un total de N CPUs a través de todos los nodos y cores, la máquina de búsqueda puede utilizar $N = T{\cdot}P$, debido a que se observo una saturación importante en nuestro cluster, cada vez que $N < T{\cdot}P$. Entonces, analizamos cuando $T>1$ para diferentes valores de P, tal que $N= T{\cdot}P$. Se debe tener presente que como $B = Q{\cdot}P$ una disminución del número de procesadores P, produce un incremento de Q por cada par (T, P), por lo tanto, es necesario establecer el valor de Q correctamente. Para completar

nuestro escenario experimental, debemos utilizar un valor real de K, el cual indica el tamaño de las listas invertidas de cada término.

Hemos observado que al ejecutar nuestra aplicación, con K=128, sin excepción para cualquier P, el mejor rendimiento se logró para un número máximo de threads OpenMP tal que N = T·P. Por lo tanto, solo mostraremos los resultados para este caso. Además y para ilustrar mejor el rendimiento entre diferentes configuraciones de (P, T) mostramos los resultados en función del tiempo máximo de ejecución alcanzado. Todas la medidas fueron ejecutadas 5 veces utilizando una secuencia de consultas extraídas desde el log de la base de datos.

Ejecutamos dos series de experimentos, en el primero se muestra el caso con un alto tráfico de consultas B=512, y como segundo caso, un tráfico moderado de consultas B=128. Hemos llamado a estos experimentos `run-A` y `run-B` respectivamente, los cuales se ejecutaron en dos nodos del cluster. También, se estudio lo ocurrido en un solo nodo, para lo cual se redujo el tráfico de consultas a la mitad B/2 y, por lo tanto, tenemos los casos homólogos llamados `run-C` y `run-D`.

Además se estudiaron dos tipos de algoritmos de clasificación de consultas, uno denominado "light ranking" en que los documentos se clasifican utilizando una pequeña cantidad del tiempo de cálculo, y el otro llamado "heavy ranking" el cual requiere mucha más computación.

5.2.3. Resultados

En este punto se intenta dar respuesta a las preguntas formuladas antes acerca de la configuración más adecuada para una máquina búsqueda en la Web en una arquitectura CMPs. Primero, analizaremos resultados de rendimiento obtenidos en un único core del cluster que incluye dos procesadores multicores Intel Quad-Xeon.

Las tablas 5.1.a y 5.1.b muestran la comparación en términos de rendimiento de speedup entre dos paradigmas de programación paralela (BSP vs

OpenMP) y la combinación de ambos. Para calcular la aceleración escogimos la configuración (P, T) que entregó el peor throught, la cual corresponde a los valores con 1.0 en la tabla. Los resultados son bastante satisfactorios bajo este caso, aunque se puede observar para un tráfico moderado de consultas, se produce un importante desequilibrio en la carga de trabajo. Aunque el speedup para un alto tráfico de consultas no es relevante, nos gustaría hacer inca-pie en el rendimiento que proporcionan. La tabla muestra una ganancia al paralelizar con OpenMP, aunque tal paralelización solo se aplica en la etapa del ranking de los documentos. Es importante destacar, que la mejor configuración paralela en un nodo, corresponde con T=8, en comparación con cualquier combinación posible para paso de mensajes. La gran diferencia entre ambos paradigmas de programación, tiene relación con la sobrecarga asociada a la duplicidad del índice invertido y la saturación de los accesos a memoria/disco en el enfoque de BSP. Esta desventaja se puede evitar utilizando OpenMP; los efectos positivos que proporciona, tienen relación a la jerarquía de memoria entre los T threads, que puede generar la memoria-prefetching y una reducción importante del cuello de botella.

(a) **Heavy Ranking Un nodo**

P	T	Runs-C	Runs-D
1	8	1.04	1.16
2	4	1.04	1.16
4	2	1.04	1.12
8	1	1.00	1.00

(b) **Light Ranking Un nodo**

P	T	Runs-C	Runs-D
1	8	1.09	1.40
2	4	1.10	1.39
4	2	1.10	1.29
8	1	1.00	1.00

(c) **Heavy Ranking Dos nodos**

P	T	Runs-A	Runs-B
2	8	1.29	1,76
4	4	1.27	1,72
8	2	1.22	1,54
16	1	1.00	1,00

(d) **Light Ranking Dos nodos**

P	T	Runs-A	Runs-B
2	8	1.48	2.15
4	4	1.44	2.08
8	2	1.35	1.76
16	1	1.00	1.00

Cuadro 5.1: Mejora de los tiempos de respuesta.

En las tablas 5.1.c y 5.1.d mostramos el speedup logrado en 2 nodos CMPs, considerando como base un bajo throughtput. Como se puede observar, nuestro sistema paralelo propuesto supera a cualquier configuración donde se combinan P y T threads. Este fenómeno se debe principalmente al incremento de las comunicaciones entre los P procesos y la interfaz de comunicación compartida. Este efecto no es tan relevante en un solo nodo, porque las comunicaciones se realizan a través de la memoria compartida. Por ejemplo, con la mejor configuración (T=8) en `run-A` y `run-B`, existe un deterioro apreciable en términos de speedups debido a los efectos de comunicación (en torno al 10-35 %). Sin embargo, en tales situaciones y la tendencia lo confirma, nuestro esquema de paralelización híbrida obtiene el mejor throughput en comparación con sus contra partes bajo cualquier tasa de tráfico y tipo de ranking, lo cual se refleja en mejoras entre 1.35 y 2.15.

También es interesante estudiar el comportamiento de diferentes configuraciones cuando no todos los cores están disponibles. La figura 5.1 muestra el speedup alcanzado en la fase *heavy ranking* para dos nodos con alto y moderado tráfico de consultas `[Speedup = time(P=2,T=1)/time(P,T)]` . Como era de esperar, nuestro sistema paralelo basado en OpenMP presenta mejores resultados para un número de threads asignados. Estos resultados cercanos al óptimo abren la posibilidad de seguir introduciendo threads OpenMP a otras partes del código, inicialmente de bajo peso en el tiempo total de ejecución, tales como el fetching de listas invertidas.

Aunque nuestro estudio se ha visto limitado por la disponibilidad de dos nodos, que incluyen CMPs, hemos comprobado la escalabilidad de la aplicación de este método en un grupo más grande con el fin de extrapolar los resultados a un escenario más realista. Hemos sido capaces de demostrar que es posible lograr eficiencias cercanas a 85 % en un sistema con 32 nodos. `[(Eficiencia = Tserie/Tparallelo*numero de nodos]`

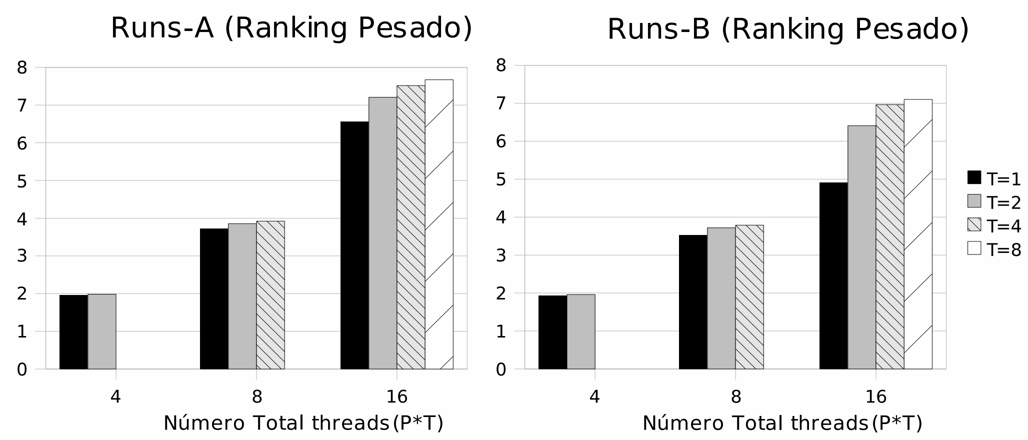


Figura 5.1: Speedup para dos nodos CMPs para un tráfico alto/bajo de consultas

Capítulo 6

Conclusiones

El objetivo de este trabajo fue proponer técnicas para mejorar el rendimiento de las Máquinas de Búsqueda en la Web, basados en el uso eficiente del paralelismo procedente del hardware de multithreading disponible en los CMPs. Nuestros resultados muestran un incremento significativo del rendimiento para un gran flujo de consultas.

Las actuales máquinas de búsqueda son completamente asíncronas y presentan problemas de serializaci'on en secciones críticas de la aplicación. En general, estas máquinas utilizan técnicas para evitar recuperar la lista completa de documentos y, por lo tanto, tienen escencialmente las mismas dificultades para obtener las ventajas que proporcionan los sistemas CMPs.

Dado este enfoque, planteamos dos escenarios, donde se trato de explotar al máximo el nivel de paralelismo de los CMPs. En un primer escenario, se mostró lo ocurrido para un alto tráfico de consultas que llegan al buscador. En este caso en particular, la máquina de búsqueda trabajo de manera síncrona; tratamos de mejorar la cooperación entre threads con datos distribuidos y threads que atienen a grupos de consultas con caraterísticas similares (términos similares). Los resultados obtenidos muestran que el speedup aumenta proporcionalmente con el número de threads, y en el escenario más

favorable, nuestra máquina de búsqueda síncrona alcanza un speedup de 22 utilizando 32 threads.

Por otro lado, mostramos lo ocurrido con un buscador que opera de modo asíncrono, es decir, el tráfico de consultas, es menor que en el caso anterior. Los resultados que hemos obtenido en este punto, nos indican que la ganancia procedente del paralelismo, no es realmente signficativa; principalmente porque en cada iteración de una consulta, la cantidad de información que se procesa, es un muy pequeña.

En ambas estrategias, utilizamos directivas de OpenMP para el majeno interno dentro de una consulta a través de threads y la comunicación vía BSP.

Por otra parte, dados los resultados obtenidos, intuitivamente se podría pensar que la mejor manera de explotar el nuevo nivel de paralelismo disponible en CMPs, es ampliar el número de procesos dedicados al procesamiento de consultas. Sin embargo, esto implica una división del índice invertido a través de los cores que pueden causar serios conflictos para los recursos compartidos, especialmente para la memoria.

Nuestra propuesta trata de minimizar estos efectos mediante la aplicación de paralelismo híbrido de una manera tal que permite a los threads OpenMP realizar su trabajo sin interferencias. Hemos mostrado que esta combinación puede ser muy efectiva en aumentar el throughput, bajo situaciones de tráfico alto y moderado de consultas de usuarios.

Apéndice A

En esta sección se detallan las características técnicas de las plataformas donde fueron ejecutados los algoritmos de esta tesis; las métricas de desempeño utilizadas y la validación de la base de datos utilizada en cada experimento.

A.1. Plataforma experimental

Trabajamos con dos tipos de plataformas experimentales. La primera de ellas, utilizada en el capítulo 4, fue un procesador UltraSPARC T1 de Sun Microsystem, cuyas características más importantes se encuentran resumidas en la siguiente tabla A.1. Por otro lado, trabajamos en un cluster de memoria compartida con Linux. Cada nodo incluye dos procesadores multicores Intel Quad-Xeon. Sus principales características se encuentran resumidas en el cuadro A.2. Esta tecnología CMPs, la utilizamos en el capítulo 5.

A.2. Medidas de desempeño

Dentro de las medidas de desempeño que fueron realizadas, se tienen:

Throughput evalúa el rendimiento del sistema con respecto a las consultas que son procesadas en cada superstep.

<table>
<tr><td rowspan="3">Processor</td><td colspan="2">SUN UltraSPARC-T1 8 core processor (1.2GHz)
(4-way fine-grain multithreading core)</td></tr>
<tr><td>L1 Cache
(per core)</td><td>16+8 KB (instruction+data)
4-way associative, LRU</td></tr>
<tr><td>L2 Unified
Cache</td><td>3MB (4Banksx768KB)
12-way associative, pseudo-LRU</td></tr>
<tr><td>Memory</td><td colspan="2">16 GBytes
(4x4GBytes) DIMMS
533 MHz DDR2 SDRAM</td></tr>
<tr><td>Operating System</td><td colspan="2">SunOS 5.10 (Solaris 10) for UltraSparcT1</td></tr>
<tr><td>Sun C/C++ Compiler
v5.8 Switches</td><td colspan="2">-fast -xarch=v9 -xipo=2
Parallelization with OpenMP: -xopenmp=parallel</td></tr>
</table>

Cuadro A.1: Principales característas del procesador UltraSPARC T1

<table>
<tr><td rowspan="3">Procesador</td><td colspan="2">Intel Quad-Xeon (2.66 GHz)</td></tr>
<tr><td>L1 Cache
(per core)</td><td>4x32KB + 4x32KB (inst.+data)
8-way associative, 64 byte per line</td></tr>
<tr><td>L2 Unified
Cache</td><td>2x4MB (4MB shared per 2 procs)
16-way associative, 64-byte per line</td></tr>
<tr><td>Memoria</td><td colspan="2">16 GBytes
(4x4GB) 667 MHz FB-DIMM memory
1333 MHz system bus</td></tr>
<tr><td>Sistema
Operativo</td><td colspan="2">GNU Debian System Linux
kernel 2.6.22-SMP for 64 bits</td></tr>
<tr><td>Intel C/C++ Compilador
v10.1 Switches (icc)</td><td colspan="2">-O3 -march=pentium4 -xW -ip -ipo
Parallelization with OpenMP: -openmp</td></tr>
<tr><td>Librería MPI</td><td colspan="2">mpich2 v1.0.7 compiled with icc v10.1</td></tr>
<tr><td>Librería BSPonMPI</td><td colspan="2"><code>http://bsponmpi.sourceforge.net</code></td></tr>
</table>

Cuadro A.2: Principales características Intel Quad-Xeon

Tiempo de ejecución calcula el tiempo transcurrido para procesar un conjunto de Q consultas utilizando P procesadores. La medición comienza una vez que han sido leídos todos los archivos y estos quedan almacenados en la RAM de cada nodo.

Todas las estrategias fueron implementadas bajo la biblioteca de comunicación `BSPonMPI`, es decir, los algoritmos fueron escritos utilizando el estándar `BSP`, pero las primitivas de comunicación fueron sobre `MPI`.

A.3. Validación de las base de datos

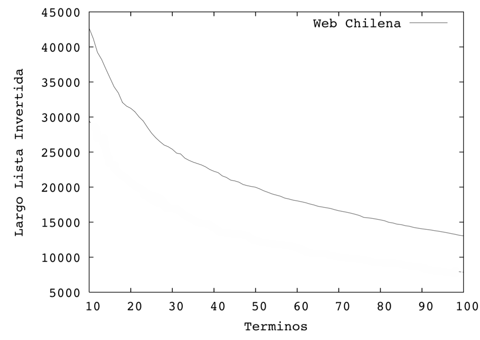
Se ha dispuesto de una base de datos de tipo texto, *Web Chilena* con texto es castellano con un tamaño de 2 Gb. de texto.

La ley de Zipf dice que: si en un texto hay R palabras distintas y las ordenamos de mayor a menor frecuencia, entonces la frecuencia Y de una palabra es proporcional a 1/X, siendo X el rango de la palabra en la lista. La constante de proporcionalidad depende de R.

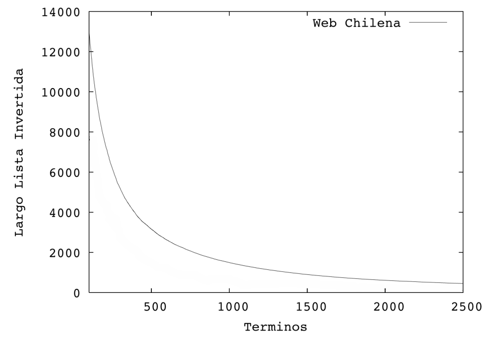
Que una medida cumpla la Ley de Zipf es bueno desde el punto de vista del ranking, ya que hay pocos elementos de buena calidad, pero es malo desde el punto de vista de procesar la información: existen muchos elementos de baja calidad que también deben ser indexados. En este caso, esta afirmación se cumple para la base de datos utilizada.

La figura A.1 muestra por cada término el tamaño de sus listas invertidas. En A.1.a se grafican los primeros 100 términos y en A.1.b los siguientes términos de la base de datos.

Por otro lado, la figura A.2 muestra el histograma de frecuencias de cada término en el log de consultas, es decir, la cantidad de veces que un término se encuentra en el log. A.2.a gráfica los primeros 100 términos de la bases de datos, y A.2.b los siguientes.

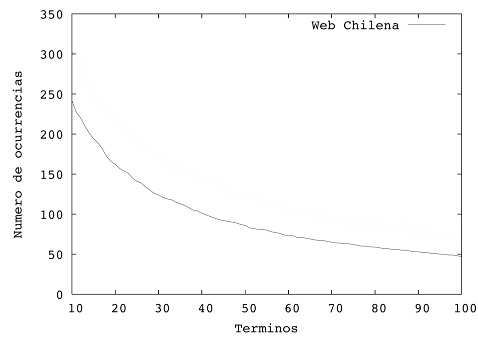


(a)

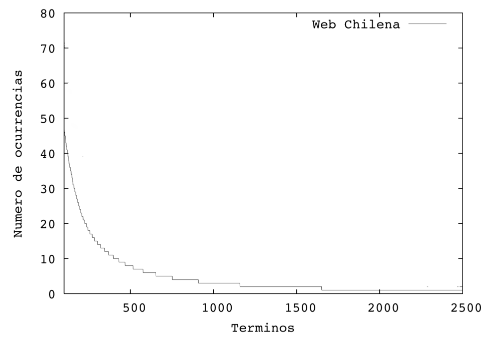


(b)

Figura A.1: Largo de la lista invertida de cada término

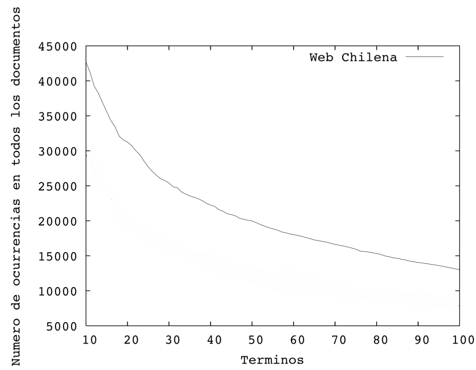


(a)

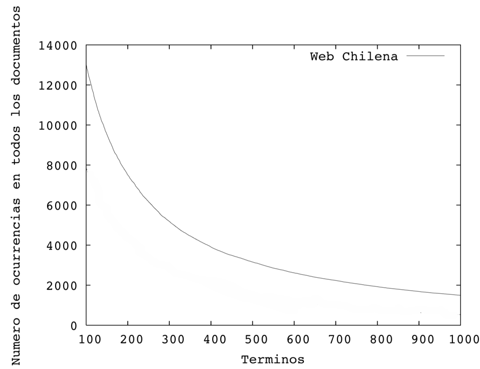


(b)

Figura A.2: Histograma de frecuencias de los términos en el log de consultas



(a)



(b)

Figura A.3: Frecuencia de cada término en todos los documentos

Finalmente en figura A.3 muestra la frecuencia de cada término en todos los documentos que conforman la base de datos. Al igual que en la figuras anteriores de esta sección, A.3.a muestra el comportamiento de los 100 primero términos, y A.3.b el de los siguientes términos.

Bibliografía

[1] P. Abad, V. Puente, P. Prieto, and J.A. Gregorio. Rotary router: An efficient architecture for cmp interconnection networks. *34th International Symposium on Computer Architecture (ISCA). San Diego, CA, USA*, June 2007.

[2] V.N. Anh and A. Moffat. Random access compressed inverted files. *9th Australasian Database Conference, Perth, Australia*, pages 1–12, Feb 1998.

[3] V.N. Anh and A. Moffat. Integrated impacts for web retrieval. *Proc. 2003 Australian Document Computing Symposium, Canberra*, pages 25–30, Dec 2003.

[4] A. Arusu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Trans.*, 1(1):2–43, 2001.

[5] C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Eighth Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 10–20. (IEEE CS Press), Nov. 2001.

[6] C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. *Eighth Symposium on String Processing and Information Retrieval (SPIRE 01)*, pages 10–20, 2001.

[7] R. Baeza-Yates. Text retrieval: theory and practice. *Proceedings of the 12th IFIP World Computer Congress, North-Holland, Madrid, Spain*, pages 465–476, 1992.

[8] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ISBN:0-201-39829-X, 1999.

[9] R. Baeza-Yates, F. Saint-Jean, and C. Castillo. Web structure, dynamics and page quality. In *Proceedings of String Processing and Information Retrieval (SPIRE)*, pages 117 – 132, Lisbon, Portugal, 2002. Springer LNCS.

[10] R. Baeza-Yates and R. Schott. Parallel searching in the plane. *Proc. 9th Conference on Shape Recognition and Artificial Intelligence, Paris*, pages 557–566, 1994.

[11] A. Barroso, J. Dean, and U. H. Olzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):2002, 22-28.

[12] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. *Journal of computer and System Sciences*, 51:359–366, Dec 1995.

[13] A. Bestavros and D. Spartiotis. Probabilistic job scheduling for distributed real-time applications. *IEEE Workshop on real-tiem applications*, pages 97–102, May 1993.

[14] A. Bhattacharje, K. Ravindranathh, R. Mall, and A. Pall. A distributed dynamic real-time scheduling algorithm. *Special issue of parallel and distributed computing practices journal*, 1, Dic 1998.

[15] C. Bonacic, C. Garcia, M. Marin, M. Prieto, and F. Tirado. Improving search engines performance on multithreading processors. *8th International Meeting on High Performance Computing for Computational Science. VECPAR 2008. Toulouse, Francia*, Jun. 24-27 2008.

[16] C. Bonacic, M. Marin, C. Garcia, M. Prieto, and F. Tirado. Exploiting hybrid parallelism in web search engines.

[17] J. Callan. Distributed information retrieval. *In W.B. Croft, editor, Advances in information retrieval, Kluwer Academic Publishers*, 5:127–150, 2000.

[18] C. Castillo, M. Marin, A. Rodriguez, and R. Baeza-Yates. Scheduling algorithms for web crawling. *2nd Latin American Web Congress (IEEE-CS)*, Oct 2004.

[19] S.H. Chung, H.C. Kwon, K.R. Ryu, H.K. Jang, J.H. Kim, and C.A. Choi. Parallel information retrieval on a sci-based pc-now. In *Workshop on Personal Computers based Networks of Workstations (PC-NOW 2000)*. (Springer-Verlag), May 2000.

[20] G. V. Cormack, C. R. Palmer, and L. A. Clarke. Efficient construction of large test collections. *Research and Development in Information Retrieval*, pages 282–289, 1998.

[21] Intel Corpation. Intel C/C++ and Intel Fortran Compilers for Linux. Available at http://www.intel.com/software/products/compilers.

[22] G.V. Costa. Estrategias de búsqueda paralela para un servidor web. *Tesis Maestría, Universidad Nacional de San Luis, Argentina*, 2006.

[23] G.V. Costa and Printista M. Modelización bsp de listas invertidas paralelas. *Workshop Chileno de Sistemas Distribuidos y Paralelismo*, Nov 2004.

[24] G.V. Costa, M. Printista, and M. Marin. A parallel search engine with bsp. *Third Latin American Web Congress (LA-Web 2005), Buenos Aires, Argentina*, Oct 2005.

[25] G.V. Costa, M. Printista, and M. Marin. Improving web searches with distributed buckets structures. *4th Latin American Web Congress, Puebla, Mexico IEEE-CS*, pages 119–126, Oct 2006.

[26] O. de Kretser, A. Moffat, T. Shimmin, and J. Zobel. Methodologies for distributed information retrieval. *18th International Conference on Distributed Computing Systems, Amsterdam*, pages 66–73, May 1998.

[27] D. Florescu, A.Y. Levy, and A.O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[28] S. Garcia, H. Williams, and A. Cannane. Access-ordered indexes. *Proceedings of the 27th Conference on Australasian Computer Science Dunedin, New Zealand*, 26:7–14, 2004.

[29] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the pvm system. *Concurrency: Practice and Experience*, 4:293–311, April 1992.

[30] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1996.

[31] W. Gropp and E. Lusk. Pvm and mpi are completely different.

[32] J. R. Haritsa and S. Seshadri. Real-time index concurrency control. *Real-Time Database Systems. http://citeseer.ist.psu.edu/haritsa00realtime.html*, pages 59–74, 2001.

[33] D. Harman. *Relevance Feedback and Others Query Modification Techniques*. en Information retrieval: data structures and algorithms, 1992.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[35] http://bsponmpi.sourceforge.net/. The bsp on mpi communication library.

[36] http://www.bsp worldwide.org. Bsp and worldwide standard.

[37] http://www.csm.ornl.gov/pvm. Parallel virtual machine.

[38] http://www.mcs.anl.gov/mpi/index.html. Message passing interface.

[39] http://www.netlib.org/pvm3/book/pvm book.html. Parallel virtual machine.

[40] http://www.uni paderborn.de/bsp. Bsp pub library at paderborn university.

[41] B.S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[42] P. Kongetira, K. Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[43] A.A. MacFarlane, J.A. McCann, and S.E. Robertson. Parallel search using partitioned inverted files. In *7th International Symposium on String Processing and Information Retrieval*, pages 209–220. (IEEE CS Press), 2000.

[44] G. Manimaran and C. Silva Ram Murthy. A efficient dymanic scheduling algorithm for multiprocessor real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 9:312–319, Mar 1997.

[45] M. Marin. Parallel text query processing using composite inverted lists. *Second International Conference on Hybrid Intelligent Systems (Invited session on Web Computing)*, Dec. 2002.

[46] M. Marin, C. Bonacic, G.V. Costa, and C. Gomez. A search engine accepting on-line updates. *13th European Conference on Parallel and Distributed Computing (EuroPar 2007), IRISA,Rennes, France*, pages 28–31, Aug 2007.

[47] M. Marin, C. Bonacic, V. Gil-Costa, and C. Gomez. A search engine accepting on-line updates. In *Euro-Par '07: 13th International Conference on Parallel and Distributed Computing*, pages 348–357, 2007. LNCS 4641.

[48] M. Marin and GV Costa. High Performance Distributed Inverted Files. *In* CIKM *2007, Nov. 6-9, 2007, Lisboa, Portugal*, 2007.

[49] M. Marin and G.V. Costa. (Sync|Async)$^{+}$ MPI Search Engines. *In EuroPVM/MPI 2007, Paris, France, Oct. 2007*, 2007.

[50] Sun Microsystem. Opensparc. world's first free 64-bit cmt microprocessors. http://www.opensparc.net/.

[51] W. Moffat, J. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval, published on-line*, 5:2006, October.

[52] J. H. Moreno. Chip-level integration: the new frontier for microprocessor architecture. *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures. Cambridge, Massachusetts, USA*, pages 328–328, 2006.

[53] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:19–27, 2001.

[54] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Number 3 in Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.

[55] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[56] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[57] B. Ribeiro-Neto, J. Kitajima, G. Navarro, C. Santana, and N. Ziviani. Parallel generation of inverted lists for distributed text collections. In *XVIII Conference of the Chilean Computer Science Society*, pages 149–157. (IEEE CS Press), 1998.

[58] B.A. Ribeiro-Neto and R.A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Third ACM Conference on Digital Libraries*, pages 182–190. (ACM Press), 1998.

[59] S. Robertson and K. Jones. Simple proven approaches to text retrieval. *Tech. Rep. TR356, Cambridge University Computer Laboratory*, 1997.

[60] P.C. Saraiva, E.S. Moura, N. Ziviani, R. Fonseca, W. Meira, C. Murta, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *24th ACM SIGIR Conference*. (IEEE CS Press), Sept. 2001.

[61] D. Sheahan. Developing and tuning applications on ultrasparc t1 chip multithreading systems. Technical report, Sun Microsystems. Sun Blue-Prints Online, October 2007.

[62] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Computing Laboratory, Oxford University, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[63] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[64] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[65] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, pages 1–56, Jul 2006.

[66] J. Zobel, A. Moffat, and K.Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, pages 453–490, Dec 1998.