Universidad de las Ciencias Informáticas

Facultad 6

Algoritmos para la ejecución paralela de consultas de PostgreSQL en computadores multinúcleos y/o multiprocesadores

Trabajo de Diploma para optar por el título de
Ingeniero en Ciencias Informáticas

Autores: Osmel Barreras Piñera
Frank A. Rodriguez Solana

Tutores: DrC. Liesner Acevedo Martínez
Ing. Adrián Quintero Henríquez

Junio 2012

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de ________ del año ________.

Osmel Barreras Piñera	Frank A. Rodriguez Solana
________________	________________
Firma del Autor	Firma del Autor

DrC. Liesner Acevedo Martínez	Ing. Adrián Quintero Henríquez
________________	________________
Firma del Tutor	Firma del tutor

DATOS DE CONTACTO

Autores:

Osmel Barreras Piñera.

Universidad de las Ciencias Informáticas, 2012

Correo Electrónico: obarreras@estudiantes.uci.cu

Frank A. Rodriguez Solana

Universidad de las Ciencias Informáticas, 2012

Correo Electrónico: fasolana@estudiantes.uci.cu

Tutores:

DrC. Liesner Acevedo Martínez.

Correo Electrónico: frodo@uci.cu

Universidad de Ciencias Informáticas, 2012

Categoría Docente: Auxiliar

Ing. Adrián Quintero Henríquez.

Correo Electrónico: aquintero@uci.cu

Universidad de Ciencias Informáticas, 2012

Categoría Docente: Instructor recién graduado

AGRADECIMIENTOS

A mi familia toda, por compartir los momentos que han hecho de mí una persona preparada para enfrentarme a la vida y poder cumplir todos mis sueños y metas.
A mi amigo y dúo de tesis, por compartir sus conocimientos y permitirme trabajar con él durante estos meses de desarrollo productivo e incansable.
A mis tutores Liesner y Adrián por contribuir a mi formación profesional y personal, y por su constante preocupación y por su apoyo durante este largo proceso de trabajo científico e investigativo.
Muy en especial a mi churri por estar junto a mí en todo momento, por su apoyo incondicional, por compartir alegrías y tristezas, por ser parte de mis sueños y ayudarme a realizarlos.
A todas las personas que han contribuido con mi formación profesional, profesores y amigos les agradezco sinceramente, deseando estar a la altura de las enseñanzas y principios que compartieron conmigo.
A la revolución cubana por pagar mis estudios y permitirme una formación excelente y gratuita en estos cinco largos años que he cursado en esta universidad.

Osmel

A toda mi familia, por guiarme en el transcurso de mi formación como profesional, y apoyarme incondicionalmente, en todo momento.

A mi abuela Geraldina, mi tía Gloria y mi tío Joseito, por ser mis padres en una etapa importante de mi vida.

A Jose, mi padrastro, por estar siempre presente para mi mamá, en las buenas y en las malas. Y por compartir con ella la dura tarea de formar a su hijo.

A todos mis amigos, los que me han acompañado a lo largo de estos cinco años de estudio dentro y fuera de la universidad.

A mis tutores Adrian y Liesner, por dedicar parte de su tiempo a la formación profesional de estos dos locos. Gracias a los dos, por inspirarnos, y por hacer de nosotros par de profesionales.

A mi amigo y dúo de tesis, por comprender mi nivel de abstracción. En especial a su mamá, su papá, su tía Anita y a su hermano, gracias a todos ustedes por acogerme y hacerme sentir parte de su familia.

A mi amigo Yonnis, por saber escucharme, por comprenderme y aconsejarme, y por estar presente en mis buenos y malos momentos durante estos cinco años.

A mi amigo José Lazaro, por molestar cuando hace falta, y por creer en mí.

A Yaimirys, por educarme en muchas cuestiones de la vida, y hacerme entender un poquito mejor a las personas. Gracias, por darle sentido a mi vida, y por lograr que sea yo mismo cuando estoy a su lado.

Frank Alberto

DEDICATORIA

A mi mamá, mi papá y a mi tía Anita por los tantos años de sus vidas que han dedicado a mi formación y al apoyo de todas mis decisiones con sus consejos y enseñanzas.

A mi hermano por ser parte de mi vida y acompañarme en los buenos y malos momentos.

A mis amigos y a las personas que han influido positivamente en mi desarrollo como profesional y me han ayudado a lograr todas las metas que me he propuesto en el transcurso de mi paso por la vida universitaria.

Osmel

A mi abuelo Gerardo, doquiera se encuentre, siempre ha estado presente en mi vida y ha sido mi apoyo para alcanzar mis metas.

A mi mamá, por dedicar su vida a mí, por hacerme quien soy y tratar de hacer de mí una mejor persona.

A mi papá, que aunque todos resalten sus defectos, de él llevo la sangre y de él heredé virtudes y defectos.

A toda mi familia, por su apoyo y consejos, a lo largo de mi vida, para ayudarme a alcanzar mis metas.

A mis amigos, por sus influencias en mi desarrollo como profesional.

Frank Alberto

RESUMEN

PostgreSQL es uno de los gestores de bases de datos, de software libre, más utilizados en la actualidad. El mecanismo de ejecución de una consulta simple en PostgreSQL no aprovecha las capacidades de cómputo disponibles en las arquitecturas de computadoras actuales. Esto implica que los tiempos de ejecución de las consultas complejas sobre bases de datos grandes sean altos. Las soluciones computacionales de altas prestaciones, están orientadas al desarrollo de algoritmos paralelos que aprovechen el mayor porciento posible de los recursos de cómputo disponibles de las arquitecturas multinúcleo y/o multiprocesadores. El presente trabajo se planteó como objetivo principal desarrollar algoritmos paralelos que puedan ser utilizados en el gestor de bases de datos PostgreSQL para la ejecución de consultas. Para lograrlo se analizaron los principios de la programación paralela, los distintos modelos de implementación basados en hilos y algunas de las bibliotecas de C/C++ utilizadas para su gestión. Se realizó un estudio del código fuente del gestor para evaluar la distribución del trabajo y los procesos posibles a paralelizar. Se obtuvo como resultado principal un algoritmo paralelo que sustituye el mecanismo usual (secuencial) que usa PostgreSQL para el procesamiento de tuplas en una consulta. Aplicando un conjunto de pruebas se obtuvieron resultados satisfactorios con respecto a los tiempos de ejecución, eficiencia y el rendimiento del gestor de bases de datos PostgreSQL en la ejecución de las consultas.

PALABRAS CLAVE: procesamiento paralelo de consultas, rendimiento, eficiencia, PostgreSQL, Pthread.

TABLA DE CONTENIDOS

INTRODUCCIÓN ... 1

Estructura de la tesis ... 2

CAPÍTULO 1: REVISIÓN BIBLIOGRÁFICA ... 4

1.1 Introducción ... 4

1.2 Arquitecturas Multinúcleos ... 4

1.2.1 Clasificación ... 4

1.3 Diferencias entre paralelismo y concurrencia ... 5

1.3.1 Modelos de Algoritmos Paralelos ... 6

1.3.2 Evaluación de los algoritmos paralelos ... 8

1.4 Elementos sobre procesos e hilos. Programación Concurrente y Multihilo ... 9

1.4.1 Modelo de Procesos ... 9

1.4.2 Estados de los procesos ... 10

1.4.3 Modelo de Hilos ... 11

1.4.4 Estados de los Hilos ... 12

1.4.5 Utilización de los Hilos ... 12

1.5 Implementaciones de modelos basados en hilos ... 13

1.5.1 Trabajo con hilos en C/C++ ... 15

1.6 Gestor de bases de datos PostgreSQL ... 17

1.6.1 Consideraciones Arquitectónicas ... 18

1.6.2 Flujo de consultas en PostgreSQL ... 19

1.6.3 El módulo ejecutor de PostgreSQL ... 20

1.6.4 Los árboles Plan y Plan de Estado ... 21

1.6.5 Administración de la memoria ... 22

1.7 Conclusiones del capítulo 1 ... 22

CAPÍTULO 2: TECNOLOGÍAS Y CARACTERÍSTICAS DE LA SOLUCIÓN ... 23

2.1 Introducción ... 23

2.2 Herramientas de software ... 23

2.2.1 Lenguaje de Programación C/C++ ... 23

2.2.2 Colección de Compiladores GCC ... 23

2.2.3 Entorno Integrado de Desarrollo Eclipse ... 23

2.2.4 PgAdmin: Herramienta de Administración del gestor de BD PostgreSQL ... 24

2.3 Herramientas de hardware ... 24

2.4 La biblioteca Pthreads ... 24

2.4.1 Gestión de hilos ... 24

2.4.2 Cerrojos ... 26

2.4.3 Semáforos ... 26

2.4.4 Monitores ... 26

2.4.5 Barreras ... 27

2.5 Control de flujo para el procesamiento de consultas en el módulo ejecutor ... 27

2.6 Fundamentos de la solución propuesta ... 31

2.7 Conclusiones del capítulo 2 ... 35

CAPÍTULO 3: IMPLEMENTACIÓN, PRUEBAS Y RESULTADOS ... 36

3.1 Introducción 36
3.2 Implementación de la solución propuesta 36
3.3 Diseño de los Casos de Prueba 42
3.3.1 Pruebas del algoritmo paralelo 42
3.3.2 Pruebas de rendimiento para el servidor PostgreSQL 43
3.4 Resultados 43
3.5 Conclusiones del capítulo 3 47
CONCLUSIONES GENERALES 49
RECOMENDACIONES 50
REFERENCIAS BIBLIOGRÁFICAS 51
BIBLIOGRAFÍA 52

INTRODUCCIÓN

Uno de los gestores de bases de datos más utilizados por la comunidad de desarrolladores de la UCI, es PostgreSQL, porque además de ser libre, agrupa la mayoría de las funcionalidades que presentan los gestores privativos, es uno de los más potentes y seguros entre los existentes a nivel mundial.

Los especialistas de la comunidad de PostgreSQL han notado que el gestor es muy eficiente para consultas de baja complejidad, pero no al procesar grandes volúmenes de información con consultas de media y alta complejidad. Una de las causas, es la naturaleza secuencial de su mecanismo de procesamiento de consultas, que impide la explotación de todas las posibilidades de hardware que le brinda el ordenador donde se utiliza.

La computación paralela se puede considerar una técnica para solucionar problemas de alta complejidad computacional. Su aplicación se ha visto incrementada, sobre todo, por la tendencia actual de creación de nuevos procesadores con tecnología de procesamiento en paralelo (procesadores multinúcleo o múltiples procesadores).

Producto de la situación antes expuesta, el problema científico que se ha declarado para la investigación es: ¿Cómo lograr que el gestor de base de datos PostgreSQL aproveche la capacidad de procesamiento de los equipos de cómputo multinúcleos y/o multiprocesadores en la ejecución de consultas?

El objeto de estudio de esta investigación: son los mecanismos para la ejecución de una consulta del gestor de base de datos PostgreSQL y el campo de acción se enmarca en la utilización de algoritmos paralelos para la ejecución de consultas usando PostgreSQL sobre una arquitectura multinúcleo y/o multiprocesador.

Con el fin de solucionar el problema, se define como objetivo general "Desarrollar algoritmos paralelos que puedan ser utilizados en el gestor de bases de datos PostgreSQL para la ejecución de consultas en arquitecturas multinúcleo y/o multiprocesador". Para el desarrollo y cumplimiento del mismo, se plantearon los siguientes objetivos específicos:

- Identificar las oportunidades de paralelización en los mecanismos de ejecución de una consulta usando el gestor de bases de datos PostgreSQL.
- Diseñar e implementar algoritmos paralelos para ejecutar consultas de PostgreSQL.
- Evaluar los algoritmos desarrollados con respecto a los indicadores de rendimiento para la programación paralela.
- Realizar pruebas de rendimiento con el gestor de consultas PostgreSQL.

Con el fin de cumplir todos los objetivos propuestos, se planteó la realización de las siguientes tareas investigativas:

- ✓ Estudio de las técnicas de paralelización y distribución de procesos en arquitecturas semejantes a la de PostgreSQL.
- ✓ Caracterización de las insuficiencias actuales en las estrategias de distribución y procesamiento de datos, utilizadas por el gestor de PostgreSQL.
- ✓ Diseño de algoritmos para aprovechar la capacidad de procesamiento multinúcleo y/o multiprocesador.
- ✓ Implementación de los algoritmos.
- ✓ Ejecución de pruebas de eficiencia y rendimiento para los algoritmos implementados.
- ✓ Comparación de los resultados entre las pruebas secuenciales y paralelas aplicadas, utilizando el gestor de bases de datos PostgreSQL.

Se propone la siguiente hipótesis: con la utilización de algoritmos paralelos en el gestor PostgreSQL para la ejecución de consultas de diversas complejidades, se puede solucionar el problema de la concentración de todo el procesamiento de datos sobre un solo núcleo. La distribución correcta y racional de la carga de procesamiento, utilizando hilos de procesamiento que se mapeen sobre los procesadores disponibles en el ordenador, facilitará y agilizará el procesamiento de los datos computacionales y la velocidad de respuesta del gestor a las peticiones del usuario.

Los resultados previstos son:

- Desarrollo de algoritmos paralelos para el aprovechamiento de la capacidad de cálculo disponible en equipos de cómputo multinúcleos y/o multiprocesadores y así optimizar los tiempos de respuestas del gestor de PostgreSQL durante la ejecución de consultas de media y alta complejidad.

Estructura de la tesis

Capítulo 1: Revisión Bibliográfica: En este capítulo se abordan los principales conceptos que serán de utilidad para tener un dominio básico sobre las arquitecturas multinúcleos, paralelización, concurrencia, procesos e hilos, y además se explica brevemente el funcionamiento y el procedimiento para la ejecución de consultas de PostgreSQL.

Capítulo 2: Tecnologías y características de la solución: En este capítulo se describen las principales tecnologías utilizadas durante el desarrollo de la investigación y la fundamentación teórica de los algoritmos paralelos desarrollados para el gestor de consultas PostgreSQL.

Capítulo3: Implementación, pruebas y resultados: En este capítulo se plantean los algoritmos de la solución propuesta, así como las pruebas diseñadas para su validación y los resultados de la investigación.

CAPÍTULO 1: REVISIÓN BIBLIOGRÁFICA

1.1 Introducción

En este capítulo, se explicarán las arquitecturas multinúcleos, sus clasificaciones, entre otras consideraciones arquitectónicas. Se ofrecerán un conjunto de conceptos fundamentales sobre hilos y procesos, los estados por los que transitan, observaciones sobre su implementación, ventajas y desventajas, y características puntuales de algunas de las librerías de C/C++ utilizadas en la programación concurrente utilizando hilos. Para completar el capítulo, se ofrece una breve descripción del gestor de bases de datos PostgreSQL.

1.2 Arquitecturas Multinúcleos

Con el avance de la tecnología y a lo largo de los años de existencia de los microprocesadores, su desempeño ha ido aumentado de forma sistemática. Los primeros procesadores alcanzaban una velocidad de procesamiento de 5 MHz (5 millones de operaciones por segundo) y en la actualidad llegan a los 3 GHz (3 mil millones de operaciones por segundo). Pero el aumento de velocidad trae consigo dos inconvenientes: el incremento del consumo energético y la generación de calor, que cada vez se hace más difícil disipar. Por otro lado, existía la posibilidad de aumentar dentro de un mismo procesador la cantidad de núcleos, surgiendo la arquitectura multinúcleo o multicore. Un núcleo, es la parte del procesador que ejecuta las instrucciones, por lo que incrementar su número significa incrementar la cantidad de trabajo que este pueda realizar simultáneamente.

1.2.1 Clasificación

Los procesadores o núcleos en un mismo ordenador pueden acceder a la memoria con velocidades iguales o diferentes. De aquí surge el término SMP *(Symmetric Multi-Processor)* para referirse a los sistemas donde los procesadores acceden a cualquier dirección con la misma velocidad, también conocidos por tener un *acceso uniforme a memoria* UMA (*Uniform Memory Acces*). Pero en todos los sistemas no ocurre lo mismo, aunque con una ínfima diferencia, una memoria puede estar más cerca de un procesador que del resto, por lo que este puede acceder a ella con mayor velocidad. A estos sistemas, se les conoce por tener un *acceso no-uniforme de memoria con coherencia de cache* (cc-NUMA, *cache-coherent Non-Uniform Memory Access*).

Convenientemente, los acrónimos UMA y cc-NUMA también representan las diferentes arquitecturas de los microprocesadores las cuales se dividen en los dos tipos siguientes:

- Arquitectura de memoria uniforme (UMA, *Uniform Memory Architecture*). Las aplicaciones verán la mitad de los accesos a memoria como accesos cortos y la otra mitad como accesos largos, así el tiempo de acceso a memoria de las aplicaciones será el promedio de acceso entre ambos extremos.
- Arquitectura de memoria no uniforme con coherencia de cache *(cc-NUMA, cache-coherent Non-Uniform Memory Architecture).* Cada núcleo o procesador, tiene una memoria local a la cual tienen un bajo costo de acceso, mientras que acceder al resto tiene un costo mayor. Se debe hacer consciente de esta característica al sistema operativo, el cual puede guiar a que las aplicaciones utilicen usualmente el acceso a la memoria local. Normalmente, los sistemas con esta característica implementan la coherencia de cache.

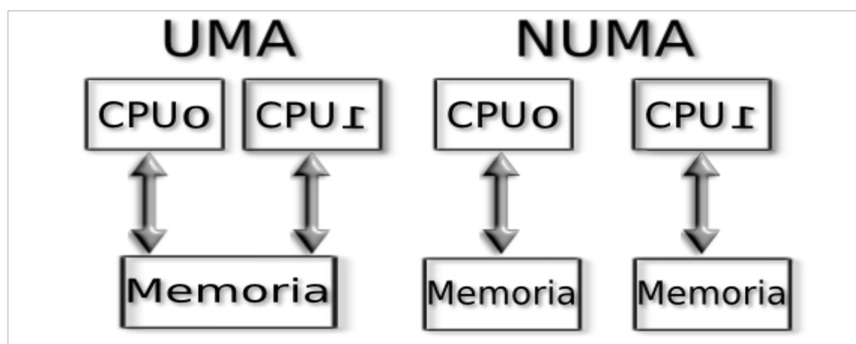

Figura 1.1 - Distribución de memoria en las arquitecturas UMA y NUMA

Estas son las dos arquitecturas de los sistemas multinúcleos y multiprocesadores, que prevalecen en el mercado. Estas arquitecturas, donde todos los núcleos o procesadores comparten la misma memoria y la tratan como un espacio global de direcciones, se conocen como arquitecturas de memoria compartida.

1.3 Diferencias entre paralelismo y concurrencia

A menudo se escuchan las palabras paralelismo y concurrencia; utilizadas erróneamente como sinónimos, por lo que se hará una simple distinción entre los dos términos.

Con el paralelismo, subprocesos simultáneos se ejecutan al mismo tiempo en varios núcleos [9]. La programación paralela, se centra en mejorar el rendimiento de las aplicaciones que utilizan una gran cantidad de energía del procesador y no se interrumpe constantemente, cuando múltiples núcleos están disponibles.

La concurrencia, es un concepto relacionado con la multitarea y las peticiones asíncronas de entrada-salida (E / S) [9]. Por lo general, se refiere a la existencia de múltiples hilos de ejecución, donde cada

uno puede obtener una porción de tiempo para ejecución antes de ser precedido por otro hilo, que también recibe una porción de tiempo para uso de la CPU. La concurrencia, es necesaria para que un programa pueda reaccionar a estímulos externos, tales como los eventos de interacción con el usuario o la entrada de dispositivos y sensores, y a su vez, que los mismos no bloqueen la ejecución de procesos que hagan uso continuo de la CPU. Comúnmente, los sistemas operativos, por su propia naturaleza, son concurrentes, incluso en un solo núcleo.

Los objetivos de la concurrencia y el paralelismo son distintos. El objetivo principal de la concurrencia, es reducir la latencia por no permitir largos períodos de tiempo sin que al menos se ejecute en la CPU alguno de los hilos desbloqueados o pendientes a ejecución. Por ejemplo, un sistema operativo con una interfaz gráfica de usuario, debe ser compatible con la concurrencia si más de una ventana a la vez puede actualizar su área de visualización en un ordenador de un solo núcleo.

El paralelismo, por otra parte, se centra solo en el rendimiento, en lograr una optimización y aprovechamiento de la posibilidad de cálculo disponible. Su objetivo es maximizar el uso del procesador en todos los núcleos disponibles, para ello, utiliza algoritmos de planificación que no son preventivos, tales como los algoritmos basados en colas de proceso o de pilas de trabajo por hacer.

1.3.1 Modelos de Algoritmos Paralelos

Un modelo de algoritmo paralelo es, normalmente, una forma de estructurar el algoritmo mediante la aplicación de principios adecuados para realizar los procesos de descomposición, asignación y aglomeración, con la premisa de minimizar las comunicaciones [2]. Un concepto a tener en cuenta es la granularidad, que es el tamaño de las piezas en que se divide una aplicación. Dichas piezas pueden ser una sentencia de código, una función, o un proceso en sí, que se ejecutarán en paralelo. La granularidad, es categorizada en paralelismo de grano fino y paralelismo de grano grueso. De grano fino, es cuando el código se divide en una gran cantidad de piezas pequeñas. Es a un nivel de sentencia donde un ciclo se divide en varios subciclos que se ejecutarán en paralelo. De grano grueso, es a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino.

- **Datos en paralelo**

El modelo datos en paralelo es uno de los modelos más simples. Las tareas son asignadas a cada uno de los procesos de forma estática o semi-estática, y cada tarea ejecuta operaciones similares con datos diferentes. El trabajo puede hacerse en fases y los datos operados en estas pueden ser diferentes. Normalmente, los datos de las fases se entremezclan con las interacciones para sincronizar

las tareas o para obtener los nuevos datos. Dado que todas las tareas deben realizar cálculos similares, la descomposición del problema, generalmente, se basa en fragmentar los datos, debido a que si se logra repartir los datos uniformemente entonces se garantiza que la carga de trabajo esté balanceada.

- **Maestro-esclavo**

El modelo maestro-esclavo, consiste en tener un proceso maestro (pueden ser varios también) el cual debe generar el trabajo y asignarlo a los procesos esclavos. Las tareas pueden asignarse a priori si se puede estimar el tamaño de las tareas, y en caso de que la asignación se haga de forma aleatoria, se debe garantizar que este balanceada la carga de trabajos. Otra forma, es atribuirles a los esclavos trabajos más pequeños en diferentes momentos. Este último, es recomendado cuando le lleva mucho tiempo al maestro crear los trabajos y por lo tanto no es conveniente que los esclavos estén esperando hasta que se hayan generado todos estos.

- **Piscina de trabajo**

El modelo piscina de trabajo o piscina de tareas, se caracteriza por realizar una asignación dinámica de tareas en procesos equilibrando la carga, en el cual cualquiera de las tareas pueda ser realizada por cualquier proceso.

La asignación puede ser centralizada o descentralizada. Los punteros de las tareas (dirección en memoria) pueden almacenarse en una lista compartida, cola de prioridad, tabla hash, un árbol o en una estructura de datos distribuida físicamente. El trabajo puede estar disponible estáticamente en el comienzo, o puede ser generado dinámicamente, es decir, los procesos pueden generar trabajos y añadirlos a la piscina global. Si el trabajo se genera de forma dinámica y se utiliza la asignación descentralizada, entonces será necesario que haya un algoritmo de detección de terminación [2] para que todos los procesos puedan saber cuándo el programa finaliza en su totalidad (por ejemplo, el agotamiento de todas las tareas posibles) y dejar de buscar más trabajo.

- **Productor-consumidor**

El modelo productor-consumidor (también conocido como tubería) representa, un flujo de datos que se transmite a través de una sucesión de procesos, donde cada uno realiza algunas tareas. Esta ejecución simultánea de diferentes programas en un flujo de datos, se llama paralelismo de flujo. Con la excepción de los procesos iniciales, la llegada de nuevos datos desencadena la ejecución de una nueva tarea por un proceso en la tubería. Los procesos pueden crear tales tuberías en forma de

matrices lineales o multidimensionales, árboles o grafos generales con o sin ciclos. Este modelo, por lo general, implica una asignación estática de las tareas en los procesos.

1.3.2 Evaluación de los algoritmos paralelos

La evaluación de los algoritmos puede hacerse teórica y experimentalmente. Para ello es necesario contar con varias métricas de evaluación de prestaciones, que consideren el tamaño de la entrada al algoritmo ***n*** y el número de procesadores ***p***. Las métricas más utilizadas son [2]:

- **Tiempo de ejecución**

Es un parámetro absoluto pues permite medir la rapidez del algoritmo sin compararlo con otro. En el caso de un programa secuencial, consiste en el tiempo transcurrido desde que se lanza su ejecución hasta que finaliza.

En el caso de un programa paralelo, el tiempo de ejecución, es el tiempo que transcurre desde el comienzo de la ejecución del programa en el sistema paralelo, hasta que el último procesador culmine su ejecución [2]. El tiempo aritmético se expresa en cantidad de FLOPS (*floating-point operations per second*), medida que indica la velocidad que tarda el procesador en realizar una operación aritmética en punto flotante.

Para sistemas paralelos con memoria distribuida, el tiempo paralelo con p procesadores, T_E, se determina de modo aproximado mediante la fórmula:

$$T_E \approx T_A + T_C - T_{SOL} \qquad \textbf{(1.1)}$$

donde T_A es el tiempo aritmético, es decir, es el tiempo que tarda el sistema multiprocesador en hacer las operaciones aritméticas; T_C es el tiempo de comunicación, o sea, el tiempo que tarda el sistema multiprocesador en ejecutar transferencias de datos; y T_{SOL} es el tiempo de solapamiento, que es el tiempo que transcurre cuando las operaciones aritméticas y de comunicaciones se realizan simultáneamente. El tiempo de solapamiento suele ser, muchas veces, imposible de calcular, por lo cual se condiciona que se realice la aproximación:

$$T_E \approx T_A + T_C \qquad \textbf{(1.2)}$$

- **Ganancia de velocidad (*Speed-Up*)**

El *Speed-Up* (S_P), para p procesadores, se determina mediante la fórmula:

$$S_P = T_S / T_P \qquad \textbf{(1.3)}$$

donde T_S es el tiempo de ejecución de un programa secuencial y T_P es el tiempo de ejecución de la versión paralela de dicho programa en p procesadores. Esta métrica indica la ganancia de velocidad que se ha obtenido con la ejecución en paralelo, respecto al algoritmo secuencial. En el mejor de los

casos, el tiempo de ejecución de un programa en paralelo con p procesadores será p veces inferior al de su ejecución en un solo procesador, teniendo todos los procesadores igual potencia de cálculo. Generalmente, el tiempo nunca se verá reducido en un orden igual a p, ya que hay que contar con las sincronizaciones y dependencias entre los procesadores.

- **Eficiencia**

La eficiencia significa el grado de aprovechamiento de los procesadores para la resolución del problema. El valor máximo que puede alcanzar es 1, que significa un 100% de aprovechamiento.

$$E = S_P / p \qquad (1.4)$$

- **Escalabilidad**

La escalabilidad es la capacidad de un determinado algoritmo de mantener sus prestaciones cuando aumenta el número de procesadores y el tamaño del problema en la misma proporción. Esta nos indica la capacidad del algoritmo de utilizar de forma efectiva un incremento en los recursos computacionales. La escalabilidad puede evaluarse mediante diferentes métricas [2]. Es conveniente tener en cuenta las características de los problemas con los que se está tratando, para elegir la métrica adecuada de escalabilidad.

1.4 Elementos sobre procesos e hilos. Programación Concurrente y Multihilo

1.4.1 Modelo de Procesos

Un proceso, consiste en una abstracción de lo que es un programa en ejecución, incluyendo los valores actuales del contador de programa, registros y variables [4].

Otra forma de definir un proceso, es como una manera de agrupar recursos relacionados, tiene un espacio de direcciones conteniendo código y datos del programa, así como otros recursos. Estos recursos pueden incluir ficheros abiertos, procesos hijos, alarmas pendientes, controladores de señales, información de contabilidad, etc. Poniendo juntos todos esos recursos en la forma de un proceso, es posible gestionarlos más fácilmente.

En este modelo, todo el software ejecutable en el ordenador, incluyendo a veces al propio sistema operativo, se organiza en un número determinado de procesos.

La rápida sustitución de un proceso a otro en algún orden, se denomina multiprogramación; en un sistema multiprogramado, la CPU también conmuta de unos programas a otros, ejecutando cada uno de ellos durante decenas o cientos de milisegundos, aunque en cualquier instante de tiempo la CPU solo está ejecutando un programa a la vez, en el transcurso de un segundo ha podido estar trabajando sobre varios programas, dando entonces a los usuarios la impresión de un cierto paralelismo.

Un proceso, es una actividad de algún tipo, tiene un programa, entrada, salida y un estado. Un único procesador puede compartirse entre varios procesos utilizando un algoritmo de planificación que determine cuándo hay que detener el trabajo sobre un proceso y pasar a atender otro diferente [4].

1.4.2 Estados de los procesos

Cada proceso es una entidad independiente, con su propio contador de programa y estado interno; los procesos necesitan a menudo interactuar con otros procesos, debido a que un proceso puede generar los datos de salida que otro proceso utiliza como entrada.

Cuando un proceso se encuentra en ejecución y se bloquea, lo hace, normalmente, porque está esperando por datos de entrada que aún no están disponibles. El tránsito hacia este estado, también es posible para un proceso que esté conceptualmente preparado, y esté parado debido a que el sistema operativo ha decidido temporalmente asignar la CPU a otro proceso.

En la figura 1.3 se puede observar un diagrama que relaciona los tres estados por los que puede transitar un proceso:

1. En ejecución (utilizando realmente la CPU en ese instante).
2. Preparado (ejecutable; detenido temporalmente para permitir que otro proceso se ejecute).
3. Bloqueado (incapaz de ejecutarse hasta que tenga lugar algún suceso externo).

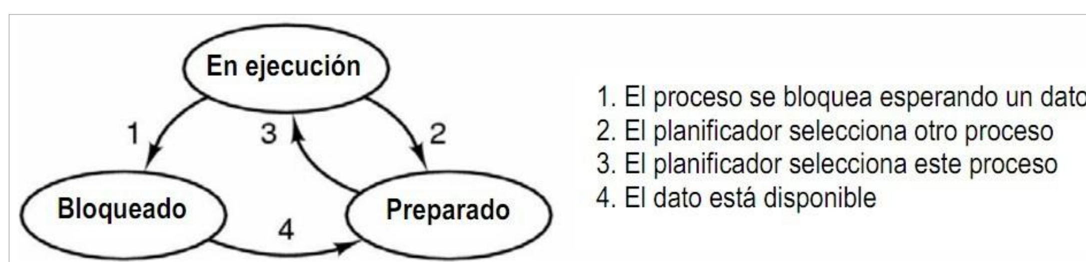

Figura 1.2 - Transiciones entre esos estados de un proceso.

Los dos primeros estados son similares, en ambos casos el proceso está dispuesto a ejecutarse, solo que en el segundo estado, temporalmente, no existe ninguna CPU disponible para él. El tercer estado es diferente a los anteriores ya que el proceso no puede ejecutarse, ni siquiera en el caso de que la CPU no tuviera ninguna otra cosa que hacer.

Utilizando el modelo de los procesos, es mucho más fácil pensar sobre lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas correspondientes a comandos tecleados por un usuario. Otros procesos son parte del sistema y desarrollan tareas tales como procesar peticiones de servicio de ficheros o gestionar los detalles del manejo de un disco o una unidad de cinta. Cuando llega una interrupción procedente del disco, el sistema toma la decisión de detener la ejecución del

proceso actual y ejecutar el proceso asociado al disco, que estaba anteriormente bloqueado esperando a que llegara esa interrupción. Así, en vez de pensar en términos de interrupciones, podemos pensar en términos de procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando tienen que esperar a que ocurra algo. Cuando el disco ha terminado de leerse, o el carácter por fin se teclea, el proceso que esperaba ese suceso se desbloquea y pasa a ser elegible para ejecutarse de nuevo.

1.4.3 Modelo de Hilos

En los sistemas operativos tradicionales, cada proceso tiene su propio espacio de direcciones y un único flujo (hilo) de control [4]. El modelo de los procesos, se basa en dos conceptos independientes: el agrupamiento de los recursos y la ejecución secuencial de un programa. A veces, es útil separar esos dos conceptos, y es aquí donde los hilos representan un papel fundamental. Frecuentemente, se desea contar con múltiples hilos de control (hilos) en el mismo espacio de direcciones ejecutándose quasi-paralelamente, como si fueran procesos separados (excepto que comparten el mismo espacio de direcciones). Aunque un hilo de procesamiento debe ejecutarse en algún proceso, el hilo y su proceso son conceptos diferentes y pueden tratarse de forma separada. Los procesos se utilizan para agrupar recursos juntos; los hilos son las entidades planificadas para su ejecución en la CPU.

El gran valor que los hilos añaden al modelo de procesos, es que permiten que haya múltiples ejecuciones en un mismo entorno determinado por un proceso, ejecuciones prácticamente independientes. La ejecución de múltiples hilos en paralelo dentro de un proceso es igual a tener múltiples procesos ejecutándose en paralelo en el mismo ordenador. Lo que se trata de conseguir con el concepto de hilo, es la capacidad de que múltiples hilos de ejecución compartan un conjunto de recursos, de manera que puedan trabajar estrechamente juntos para realizar alguna tarea. Los hilos comparten el espacio de direcciones, los ficheros abiertos y otros recursos, mientras los procesos comparten la memoria física, los discos, las impresoras y otros recursos.

Debido a que los hilos tienen algunas de las propiedades de los procesos, a veces reciben la denominación de procesos ligeros (*lightweight process*) [4]. También se utiliza el término de multihilo (*multithread*) para describir la situación en la cual se permite que haya múltiples hilos en el mismo proceso. Cuando un proceso multihilo se ejecuta sobre un sistema con una única CPU, los hilos deben hacer turnos para ejecutarse, al igual que ocurría en el modelo de procesos antes descrito. La CPU va conmutándose rápidamente de unos hilos a otros, proporcionando la ilusión de ejecución en paralelo.

1.4.4 Estados de los Hilos

Igual que un proceso tradicional (es decir un proceso con un único hilo), un hilo puede transitar por varios estados: en ejecución, bloqueado o preparado. Un hilo en ejecución tiene actualmente la CPU y está activo. Un hilo puede bloquearse esperando a que tenga lugar algún suceso externo o a que algún otro hilo lo desbloquee. Un hilo preparado está planificado para ejecutarse y lo hace tan pronto como le llega su turno. Las transiciones entre los estados de un hilo, son las mismas que las transiciones entre los estados de un proceso y ocurren como se ilustró anteriormente.

1.4.5 Utilización de los Hilos

La razón principal para usar implementaciones basadas en hilos de procesamiento durante la ejecución de tareas concurrentes, es que son numerosas las aplicaciones en las que hay varias actividades que están en marcha simultáneamente, donde la posibilidad de que alguna pueda bloquearse siempre está latente, por lo que el modelo de programación resulta más sencillo si descomponemos tal aplicación en varios hilos secuenciales que se ejecutan en paralelo. En vez de pensar en términos de interrupciones, *timers* y cambios de contexto, podemos pensar en términos de procesos paralelos. A partir de la utilización de múltiples hilos, se introduce un nuevo elemento a nuestro favor: la capacidad de las entidades paralelas para compartir entre ellas un espacio de direcciones y todos sus datos. Esta capacidad es esencial para ciertas aplicaciones, y es por lo que el tener simplemente múltiples procesos (con sus espacios de direcciones separados) no puede funcionar en estos casos [4].

Existen otras razones que se pueden citar para justificar el uso ventajoso de los hilos para la ejecución de varias tareas paralelamente. Una de ellas, es la independencia que poseen los hilos de los recursos del sistema, por lo que son más fáciles de crear y destruir que los procesos, en términos de recursos son menos costosos por lo que la creación de un hilo puede realizarse 100 veces más rápido que la creación de un proceso. Otra de las razones viene ligada al rendimiento. Los hilos no proporcionan ninguna ganancia en el rendimiento cuando todos utilizan intensamente la CPU, sino cuando hay necesidades tanto de cálculo en la CPU como de E/S, de manera que teniendo hilos se puede conseguir que esas dos actividades se solapen, acelerando la ejecución de la aplicación. También son muy útiles en sistemas que trabajen sobre arquitecturas multinúcleos, porque sí se evidencia un paralelismo auténtico[1].

[1] **Paralelismo auténtico en el sentido de que varios hilos de procesamiento pueden estar ejecutándose de forma paralela en CPU (núcleos) distintos**.

1.5 Implementaciones de modelos basados en hilos

Existen dos formas fundamentales de implementar un paquete de hilos: en el espacio del usuario, y en el núcleo (*kernel*). La elección entre esas dos implementaciones resulta moderadamente controvertida, siendo también posible una implementación híbrida. A continuación, se describen estos métodos, junto con sus ventajas y desventajas.

En espacio del Usuario

Este método consiste en poner el paquete de hilos enteramente en el espacio de usuario, lo que es muy ventajoso porque puede implementarse sobre un sistema operativo que no soporte hilos. Los hilos se ejecutan en lo alto del sistema en tiempo de ejecución (*runtime system*), que es una colección de procedimientos que gestiona los hilos, pero en consecuencia el núcleo del sistema no sabe nada de su existencia por lo que solo se gestionan procesos ordinarios con un único hilo [4].

Cuando los hilos se gestionan en el espacio del usuario, cada proceso necesita su propia tabla de hilos privada para llevar el control de sus hilos, la misma es análoga a la tabla de procesos del núcleo, salvo que sólo controla las propiedades propias de los hilos tales como el contador de programa del hilo, su puntero de pila, sus registros, su estado, etc. y es gestionada por el sistema en tiempo de ejecución.

En este tipo de conmutación de hilos, no es necesario ningún *trap*[2] al núcleo, ni un cambio de contexto, ni vaciar la memoria cache, etc., lo que representa un potente argumento a favor de los paquetes de hilos a nivel de usuario y, en términos de magnitud, hace que la planificación de los hilos sea muy rápida.

Entre otras ventajas que se pueden mencionar, está que permiten que cada proceso tenga su propio algoritmo de planificación ajustado a sus necesidades, es una ventaja adicional el no tener que preocuparse sobre si un hilo se ha quedado detenido en un momento inadecuado. Además, los hilos en el espacio del usuario se dimensionan mejor, ya que los hilos a nivel del núcleo requieren invariablemente en el núcleo algún espacio para la tabla de hilos y algún espacio de pila, lo que puede resultar un problema cuando el número de hilos es muy grande.

A pesar de su eficiencia, los paquetes de hilos a nivel de usuario tienen algunos problemas serios. Uno de ellos, surge a partir de cómo se implementan las llamadas bloqueantes al sistema; si un hilo lee desde el teclado, antes de que se haya pulsado ninguna tecla, detendría a todos los hilos del proceso, por lo es inaceptable permitir que el hilo haga realmente esa llamada al sistema. La necesidad de incorporar al sistema los hilos, aparece al permitir utilizar llamadas bloqueantes, pero sin que el bloqueo de un hilo afecte a los demás.

[2] **Trap: interrupción de software provocada por la instrucción de lenguaje máquina INT.**

Si un hilo provoca una falta de página, el núcleo, que ni siquiera sabe de la existencia de los hilos, bloquea en consecuencia al proceso entero, hasta que la E/S del disco provocada por la falta de página se complete, y eso incluso, aunque haya otros hilos ejecutables dentro del proceso.

Otro problema se manifiesta cuando un hilo comienza a ejecutarse, porque ningún otro hilo en ese proceso podrá volver a ejecutarse mientras que el primero no ceda voluntariamente la CPU. Dentro de un mismo proceso, no existen interrupciones de reloj, lo que imposibilita la planificación de los hilos para turnarse el uso de la CPU periódicamente, a menos que un hilo entre en el sistema en tiempo de ejecución por su propia voluntad; el planificador nunca tiene la oportunidad de pasar otro hilo a ejecución [4].

En espacio del núcleo (*kernel*)

Si se tratare del caso, en que el núcleo conoce y gestiona los hilos, no es necesario ningún sistema en tiempo de ejecución, ni existe ninguna tabla de hilos dentro de cada proceso. El núcleo mantiene una tabla de hilos en el sistema; cuando un hilo desea crear uno nuevo, o destruir uno que ya existe, hace una llamada al núcleo que se encarga de la creación o destrucción, actualizando la tabla de hilos del sistema [4].

La tabla de hilos del núcleo guarda los registros de cada hilo, su estado y otra información, esta es la misma que con hilos a nivel de usuario, pero almacenada en el núcleo.

Para este método, cuando se bloquea un hilo, el núcleo tiene la opción de decidir si pasa a ejecutar otro hilo del mismo proceso (si hay alguno preparado), o pasa a ejecutar un hilo de un proceso diferente; en cambio, con hilos a nivel de usuario, el sistema en tiempo de ejecución tiene que seguir ejecutando hilos de su propio proceso hasta que el núcleo le retire la CPU (o hasta que no le queden hilos preparados para ejecutarse). Todas las llamadas que puedan bloquear a un hilo en el espacio del núcleo se implementan como llamadas al sistema, a un coste considerablemente más alto que una llamada a un procedimiento del sistema en tiempo de ejecución.

Los hilos a nivel del núcleo no requieren ninguna llamada nueva de tipo no bloqueante al sistema; si un hilo de un proceso provoca una falta de página, el núcleo puede comprobar fácilmente si el proceso tiene todavía hilos ejecutables y ejecutar uno de ellos, mientras espera a que la página que provocó la falta, se cargue desde el disco. Su principal desventaja, es que el coste de una llamada al sistema es considerable, de manera que si las operaciones con hilos (creación, terminación, etc.) son frecuentes, puede incurrirse en una elevada sobrecarga para el sistema.

Híbridas

Intentando combinar las ventajas de los hilos a nivel de usuario con las ventajas de los hilos a nivel del núcleo, surgen los métodos que utilizan hilos a nivel del núcleo y multiplexan hilos a nivel de usuario sobre algunos o todos los hilos a nivel del núcleo, como se muestra en la figura siguiente.

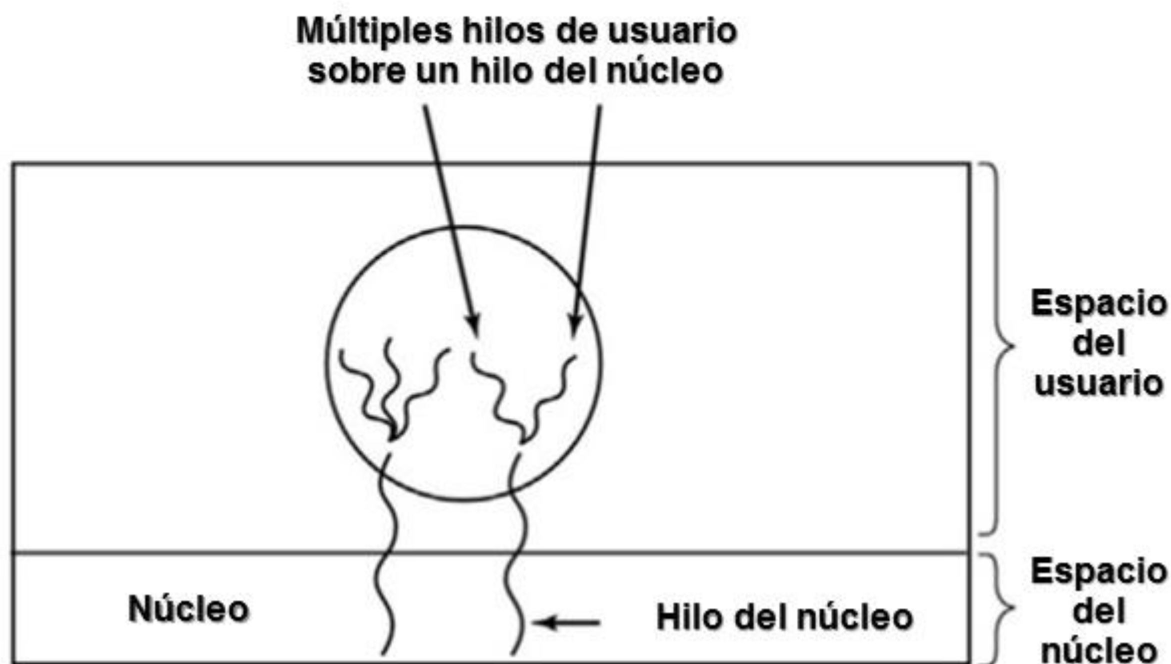

Figura 1.3 - Multiplexación de los hilos a nivel de usuario sobre los hilos a nivel de núcleo.

En este diseño, el núcleo solo tiene conocimiento de los hilos a nivel del núcleo, ocupándose de su planificación. Algunos de estos hilos pueden tener multiplexados sobre ellos múltiples hilos a nivel de usuario, los que se crean, se destruyen, y se planifican de la misma manera que los hilos a nivel de usuario de un proceso que se ejecuta sobre un sistema operativo sin capacidad multihilo. En este modelo, cada hilo a nivel de núcleo tiene algún conjunto de hilos a nivel de usuario, que lo utilizan por turnos para ejecutarse.

1.5.1 Trabajo con hilos en C/C++

Los estándares POSIX (*Portable Operating System Interface*) definidos por la IEEE[3], especifican los modelos de codificación para aplicaciones portátiles de UNIX. La mayoría de los sistemas operativos UNIX y semejantes se adhieren a las principales características de estos, por lo tanto, una aplicación de codificación de las normas será portable entre las implementaciones de UNIX y en sistemas

[3] **IEEE: Instituto de Ingenieros Eléctricos y Electrónicos.**

operativos, como *Linux* y *FreeBSD* y *Mac OS X*, que también está basado en un *kernel* tipo *Unix*. *Microsoft Windows* no implementa los estándares POSIX directamente, aunque hay algunas soluciones que permiten a los programas escritos utilizando interfaces POSIX para ejecutarse en plataformas *Windows*.

Los programas multihilo, desarrollados con las interfaces estándar POSIX, permiten que una aplicación pueda crear nuevos hilos, sincronizar y compartir datos entre hilos y procesos, debido a que proporciona un gran número de métodos de sincronización e intercambio de datos. Algunas de las bibliotecas para desarrollar implementaciones de hilos en los lenguajes C/C++ son:

Pthreads: (POXIS *Threads*): Es la biblioteca que define POSIX para el trabajo con hilos dentro de las aplicaciones. Implementa como modelo de programación, el modelo de "Memoria Compartida". Se pretende que la biblioteca *Pthreads* sea portátil, y ofrece un conjunto bastante completo de funciones para crear, finalizar, y sincronizar los hilos y para evitar que diferentes hilos traten de modificar los mismos valores y recursos, al mismo tiempo: incluye las exclusiones mutuas, las cerraduras, las variables de condición, y los semáforos. Aunque la programación de *Pthreads* es mucho más compleja que otras librerías, es la más portable de todas.

En la API[4] de programación *Pthreads*, todos los datos son compartidos, pero lógicamente distribuidos entre los hilos. El acceso a los datos compartidos a nivel global tiene que ser sincronizado de forma explícita por el usuario, el acceso a los recursos globales está protegido por un candado para que solo un hilo a la vez acceda a las actualizaciones de esta variable [8].

Marcel: Es una biblioteca de hilos que provee una interfaz compilable de POSIX y un conjunto de extensiones originales. Este también puede ser compilado para proveer compatibilidad de la interfaz binaria de aplicación ABI (*Application Binary Interface*) con hilos NTPL (*Native Thread of* POSIX *Library*) en sistemas Linux, así las aplicaciones multihilo pueden usar Marcel sin ser recompiladas. [4]

La arquitectura de Marcel fue diseñada cuidadosamente para soportar un elevado número de hilos y hacer un uso eficiente de las arquitecturas jerárquicas (Ej. procesadores multinúcleo, máquinas con arquitectura NUMA).

La característica de Marcel de poseer un planificador de hilo de dos niveles (también llamado planificador N:M) mejora el rendimiento en la ejecución de paquetes de hilos a nivel de usuario, por ser capaz de multiplexar las implementaciones multihilos a nivel de usuario, con las del nivel del núcleo permitiendo un aprovechamiento más óptimo de la capacidad de uso de la CPU de ordenadores de múltiples procesadores. A fin de evitar el bloqueo de los hilos del núcleo cuando la aplicación realiza

[4] **API: Interfaz para Programación de Aplicaciones**

una llamada bloqueante al sistema, Marcel utiliza Activaciones del Planificador, o simplemente intercepta la llamada bloqueante a nivel del símbolo del sistema.

Las bibliotecas de hilos Marcel y *Pthread*, son las que presentan mejores características para el desarrollo de esta investigación. Marcel por su capacidad de soporte para un número elevado de hilos, parece la más propicia a utilizar, pero debido a que implementa su propio planificador de hilos, puede ser que esto entorpezca el trabajo de paralelizar un programa que se ejecuta de manera secuencial. La biblioteca *Pthread* permite delegar esta tarea al propio sistema operativo, por lo que la planificación de sus hilos no representa carga de trabajo ni peligro de bloqueo para el proceso.

1.6 Gestor de bases de datos PostgreSQL

Según datos tomados del sitio de la comunidad de desarrolladores de Cuba [7], PostgreSQL es un sistema gestor de base de datos objeto-relacional, que soporta gran parte del estándar SQL, se desarrolla bajo licencia BSD[5]. Es uno de los sistemas de gestión de bases de datos de código abierto más avanzados del mundo que en sus últimas versiones posee muchas características que solo se podían ver en productos comerciales de alto calibre. PostgreSQL utiliza un modelo cliente/servidor y usa multiprocesos en vez de multihilos para garantizar la estabilidad del sistema. Se ejecuta en casi todos los principales sistemas operativos: *Linux, Unix, BSDs, Mac OS, Beos, Windows*, etc. Además se puede decir que:

- ✓ Es altamente adaptable a las necesidades del cliente, con documentación muy bien organizada, pública y libre, que permite comentarios de los usuarios de la comunidad.
- ✓ Posee comunidades muy activas, varias comunidades en castellano.
- ✓ Tiene soporte nativo para los lenguajes más populares del medio: PHP, C, C++, Perl, Python, etc.
- ✓ Soporta todas las características de una base de datos profesional (disparadores, procedimientos almacenados, funciones, secuencias, relaciones, reglas, vistas, vistas materializadas, etc.).
- ✓ Puede ser extendido por el usuario añadiendo tipos de datos, operadores, funciones agregadas, funciones ventanas y funciones recursivas, métodos de indexado y lenguajes procedurales.

[5] **BSD: (Berkeley Software Distribution), es una licencia de software libre permisiva, establece muchas menos restricciones que otras licencias como la GPL, permite el uso del código fuente en software no libre.**

Características:

- Atomicidad (Indivisible): es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.
- Consistencia: es la propiedad que asegura que solo se empieza aquello que se puede acabar. Por lo tanto, se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos.
- Aislamiento: es la propiedad que asegura que una operación no puede afectar a otras. Esto garantiza que dos transacciones sobre la misma información nunca generarán ningún tipo de error.
- Durabilidad: es la propiedad que asegura que una vez realizada la operación, esta persistirá y no se podrá deshacer aunque falle el sistema.

1.6.1 Consideraciones Arquitectónicas

Primeramente, se debe entender la arquitectura básica del sistema PostgreSQL. Una sesión de PostgreSQL consiste en los siguientes procesos cooperantes (programas):

- Un proceso de servidor, que gestiona los archivos de base de datos, acepta las conexiones a la base de datos desde las aplicaciones cliente, y realiza acciones de base de datos en nombre de los clientes. El programa de servidor de base de datos se llama *postgres*.
- El cliente del usuario (*frontend*), la aplicación que quiere llevar a cabo las operaciones de base de datos. Las aplicaciones cliente pueden ser de naturaleza muy diversa: un cliente puede ser una herramienta orientada a texto, una aplicación gráfica, un servidor web que acceda a la base de datos para mostrar las páginas web, o una herramienta de mantenimiento de bases de datos especializadas. Algunas aplicaciones de cliente se suministran con la distribución de PostgreSQL, la mayoría son desarrollados por los usuarios.

Como es típico de las aplicaciones cliente / servidor, el cliente y el servidor pueden estar en diferentes máquinas. En tal caso, la comunicación ocurre a través de una conexión de red TCP / IP.

El servidor PostgreSQL puede manejar múltiples conexiones simultáneas de diferentes clientes. Para lograr esto, se hace una llamada *forks()* del sistema, la cual crea un nuevo proceso *backend* por cada conexión para gestionar las peticiones, y el proceso *postmaster* del servidor está siempre en funcionamiento a la espera de nuevas conexiones de los clientes.

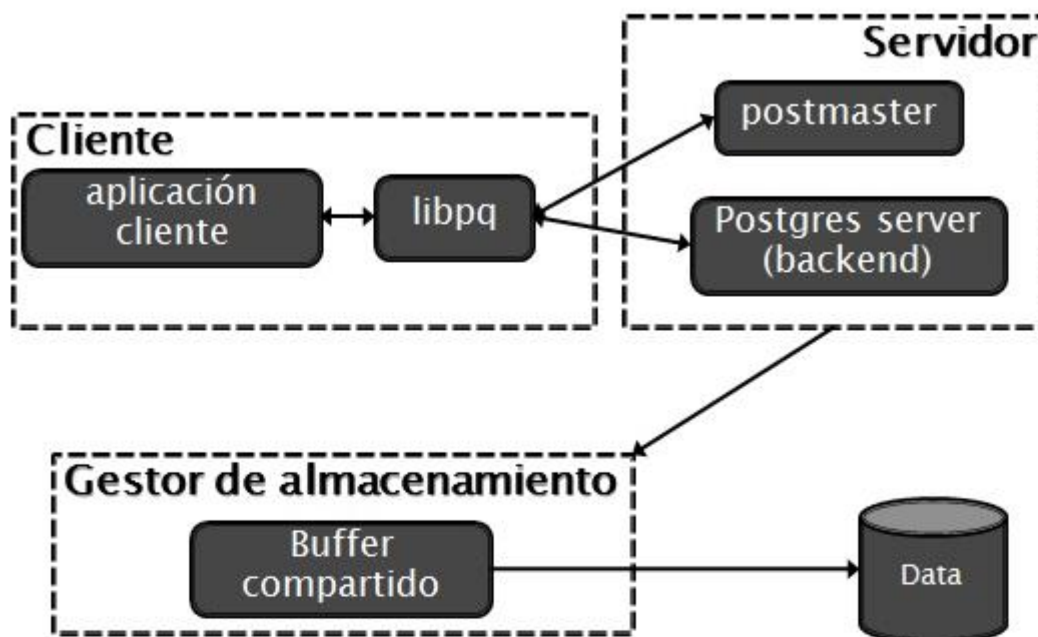

Figura 1.4 – Estructura Interna del gestor de BD PostgreSQL.

1.6.2 Flujo de consultas en PostgreSQL

Durante la ejecución de una consulta, utilizando el gestor PostgreSQL, el proceso sigue un flujo muy similar al de los compiladores tradicionales, las fases del mismo son las siguientes:

1. El programa de aplicación transmite una consulta y recibe el resultado enviado por el servidor.
2. El intérprete (*parser)* chequea la consulta enviada por el cliente, comprueba que la sintaxis es correcta y crea el árbol de la consulta.
3. El policía de tráfico (*traffic cop)* contiene al controlador principal del proceso del PostgreSQL, encargado de comunicar el resto de los módulos del servidor.
4. El sistema de reescritura toma el árbol, busca reglas almacenadas en los catálogos del sistema que pueda aplicarle y realiza las transformaciones sobre él.
5. El planeador/optimizador toma el árbol reescrito para la creación de todos los posibles caminos que conduzcan a los mismos resultados, calcula el costo de la ejecución de cada ruta para seleccionar el camino más barato, y se expande un plan de consulta completo, que será la entrada para el ejecutor.
6. El Ejecutor toma el plan de ejecución que le entrega el planeador, e inicia el procesamiento de forma recursiva, archivando, calificando y recuperando las tuplas que le serán devueltas al cliente.

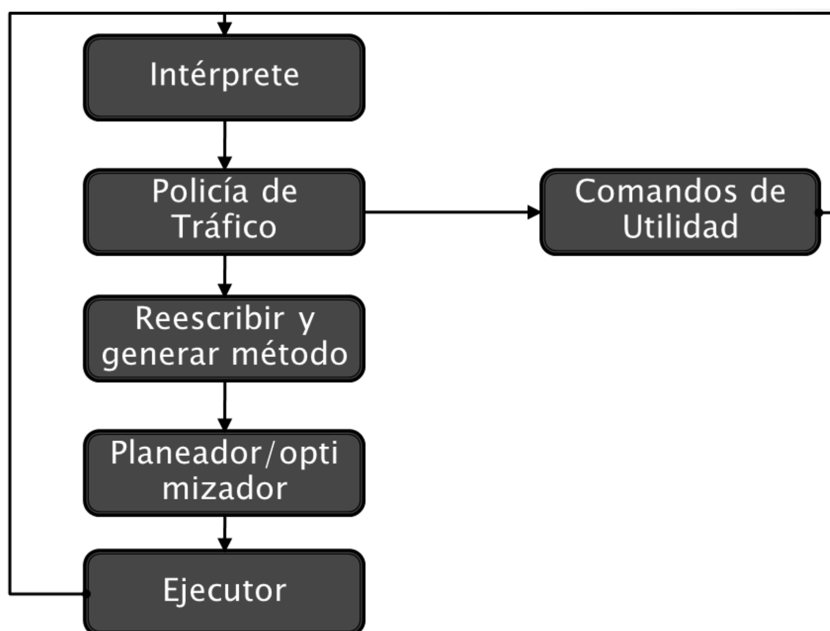

Figura 1.5- Proceso de ejecución de una Consulta en PostgreSQL.

1.6.3 El módulo ejecutor de PostgreSQL

El ejecutor, es el módulo encargado de ejecutar los procesos del árbol de nodos, el plan descriptor de una consulta lista para ejecución. En estos procesos, utilizando la biblioteca de hilos *Pthreads*, existe la posibilidad de crear un paralelismo que nos permita ganar en rendimiento y explotación de la mayor capacidad de cómputo disponible en el ordenador.

El árbol del plan es, esencialmente, una tubería de demanda de las operaciones para el procesamiento de tuplas. Cada nodo, cuando se ejecute, producirá la siguiente tupla en su secuencia de salida, o NULL si no hay más tuplas disponibles. Si el nodo no es un nodo primitivo de tipo relación-escaneo, tendrá nodos secundarios que a su vez se ejecutaran para obtener nuevas tuplas de entrada.

Algunas mejoras incluidas en este modelo básico son:

- Elección de la dirección de exploración (hacia delante o hacia atrás), para los nodos de exploración primitivos.
- Comando *Rescan* para reiniciar un nodo y hacer que genere la secuencia de salida nuevamente.
- Un esquema novedoso para evitar reexaminar los nodos innecesariamente (por ejemplo, no volver a examinar la cadena entrada si no hay cambios en los parámetros de la misma, ya que solo debe volver a leer los datos almacenados y ordenados anteriormente).

Para la ejecución de un nodo de tipo SELECT, solo es necesario entregar las tuplas resultado de nivel superior, al cliente. Si la consulta incluye una cláusula RETURNING, el nodo *ModifyTable* ofrece las filas calculadas RETURNING como salida, de lo contrario no devuelve nada.

Para la ejecución de nodos de tipo INSERT / UPDATE / DELETE, las operaciones reales de la tabla de modificación deben suceder en un nodo plan de alto nivel (*ModifyTable*).

Para la ejecución de un nodo de tipo INSERT las tuplas devueltas por el árbol del plan, *ModifyTable* se insertan en la relación-resultado adecuada.

Para la ejecución de un nodo de tipo UPDATE, el árbol del plan devuelve las tuplas calculadas para ser actualizadas, además de una "temporal"(oculta) CTID[6], columna de identificar qué filas de la tabla van a ser reemplazadas por cada tupla.

Para la ejecución de un nodo de tipo DELETE, el árbol del plan solo tiene que entregar una columna de CTID, y el Nodo *ModifyTable* visita cada una de las filas, marcándolas como filas ocultas o eliminadas.

1.6.4 Los árboles Plan y Plan de Estado

El árbol del Plan entregado por el planificador, contiene un árbol de nodos del plan (los tipos *struct* derivados del Plan de estructura). Cada nodo puede tener un Plan de árboles de expresión asociados con él, para representar a su lista de objetivos. Durante el inicio del ejecutor, se construye un árbol paralelo de idéntica estructura que contiene los nodos del estado del ejecutor, todos los planes y nodos de tipo expresión correspondientes al nodo plan de estado del ejecutor. Cada nodo en el árbol de estado tiene un puntero a su nodo correspondiente en el árbol del plan, además de los datos del estado del ejecutor, según sea necesario, para poder ejecutar ese tipo de nodo. Esta disposición permite que el árbol del plan sea de solo-lectura en lo que al ejecutor se refiere: todos los datos que sean modificados durante la ejecución se encuentran en el árbol de estado; tratando el árbol del plan como una estructura de solo-lectura, se facilita el almacenamiento en la cache y la reutilización del plan. En total, hay cuatro clases de nodos utilizados en estos árboles: los nodos del Plan, sus nodos *PlanState* correspondientes, los nodos de tipo *expr*, y sus correspondientes *ExprState* nodos; en realidad, también hay nodos de lista, que se utilizan como enlace de los cuatro tipos del árbol.

[6] **CTID: identificador que describe la ubicación física de la tupla dentro de la base de datos. Un par de números que se representan por la ctid son: el número de bloque, y el índice de tupla dentro de ese bloque.**

1.6.5 Administración de la memoria

Una consulta al contexto de memoria, se crea durante la ejecución de la función **`CreateExecutorState`**; todo el almacenamiento asignado en una invocación del ejecutor, se le asigna en ese contexto o en un contexto descendiente de ese. Esto permite la recuperación sencilla de almacenamiento durante el apagado del ejecutor, en lugar de jugar con probables fugas de almacenamiento, que solo destruyen el contexto de la memoria.

En particular, los árboles del plan de estado y los árboles de expresión de estado, se asignan en el contexto de la memoria por consulta. Para evitar las pérdidas de memoria dentro de una consulta, la mayoría del procesamiento, mientras hay una consulta en ejecución, se realiza en contextos de memoria por tupla, ya que se suelen restablecer para vaciar cada tupla, una vez procesada en memoria. Los contextos por tupla se asocian generalmente con *`ExprContexts`*, y, por lo general, cada nodo *`PlanState`* tiene su propia *`ExprContext`* para evaluar sus expresiones iguales y la lista objetivo.

1.7 Conclusiones del capítulo 1

En este capítulo se realizó un análisis sobre los conceptos fundamentales relacionados con las arquitecturas multinúcleos y sus clasificaciones; procesos e hilos, definiendo los principales conceptos y características, así como la relación que tienen con el paralelismo y la programación concurrente en los sistemas operativos y se mencionaron algunos de los principales modelos de algoritmos paralelos. Se describen los diferentes modelos de implementación de hilos y algunas de las principales bibliotecas de C/C++ para la programación multihilos, sus ventajas y desventajas fundamentales. Se realiza una breve descripción del proceso de ejecución de consultas del gestor PostgreSQL.

A pesar de no utilizar implementaciones multihilos para garantizar la estabilidad y la portabilidad del sistema, PostgreSQL podría beneficiarse de las facilidades de implementación que propone la biblioteca *Pthread* para la ejecución de modelos multihilos en sistemas de varios procesadores, permitiendo mejorar el rendimiento durante el proceso de ejecución de una consulta, y manteniendo la estabilidad del gestor al delegar la planificación de los hilos al propio sistema operativo. Se propone hacer uso de algoritmos paralelos, que permitan ejecutar las diferentes etapas del flujo de una consulta de PostgreSQL a través de hilos gestionados con la biblioteca *Pthread*.

CAPÍTULO 2: TECNOLOGÍAS Y CARACTERÍSTICAS DE LA SOLUCIÓN

2.1 Introducción

En este capítulo, se describen las tecnologías utilizadas en el desarrollo de esta investigación. Se describirá brevemente el funcionamiento de las principales funciones de la biblioteca *Pthreads* y de las rutinas más relevantes que intervienen en el flujo de control de una consulta ejecutada con PostgreSQL. Se propone un algoritmo paralelo, implementado usando hilos, para lograr aprovechar la disponibilidad de cómputo existente en el ordenador donde se encuentra instalado el servidor de bases de datos.

2.2 Herramientas de software

2.2.1 Lenguaje de Programación C/C++

Lenguaje de programación rápido y eficiente permite un control muy preciso de todo el sistema, incorpora en forma nativa herramientas para la administración de memoria y el acceso fácil al hardware sobre el que se ejecutan los programas. Son los lenguajes que con más frecuencia son usados para la creación de aplicaciones de optimización, su versión orientada a objeto C++, permite agregar un nivel más de abstracción a nuestra solución. Es un lenguaje muy eficiente, puesto que es posible utilizar sus características de bajo nivel para realizar implementaciones óptimas. A pesar de su bajo nivel, es el lenguaje más portado en existencia, habiendo compiladores para casi todos los sistemas conocidos y proporciona facilidades para realizar programas modulares y/o utilizar código o bibliotecas existentes. Existe una enorme cantidad de lenguajes modernos que se basan en C, entre ellos, C++, Visual C++, Objective C, Java, JavaScript y C#, entre otros. La similitud de estos lenguajes con el C va desde su sintaxis hasta el nombre de la gran mayoría de las funciones estándar.

2.2.2 Colección de Compiladores GCC

Colección de compiladores creados por GNU, dentro de dicha colección se encuentran compiladores para ADA, JAVA, C, C++, FORTRAN, entre otros; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina, donde ha de correr. El compilador para C y C++ es uno de los que más se acerca al estándar de C y C++, además de ser distribuido bajo la licencia GPL y LGPL.

2.2.3 Entorno Integrado de Desarrollo Eclipse

Eclipse Indigo 3.7.0, IDE (Entorno Integrado de Desarrollo) ligero y multiplataforma que facilita las tareas de edición, compilación y ejecución de programas durante su fase de desarrollo. Mediante un

arquitectura basada en *plugins* flexibles se ha convertido en un IDE genérico que se puede utilizar para desarrollar aplicaciones en diferentes lenguajes de programación como C/C++, Java, Python, Groovy, Perl [5].

2.2.4 PgAdmin: Herramienta de Administración del gestor de BD PostgreSQL

PgAdmin III 1.14.1, es una aplicación gráfica para administrar el gestor de base de datos PostgreSQL, siendo la más completa y popular con licencia de código abierto. Está escrita en C++. Además está diseñado para responder a las necesidades de todos los usuarios, desde escribir consultas SQL simples hasta desarrollar bases de datos complejas. La interfaz gráfica soporta todas las características de PostgreSQL y facilita enormemente la administración. [6]

2.3 Herramientas de hardware

En el desarrollo y las pruebas de los algoritmos, se utilizó un ordenador de escritorio que dispone de 2 procesadores Intel(R) Core(TM) 2 Duo CPU E4500 2.20GHz, con memoria RAM de 1 GB DDR-2, 250 GB.

2.4 La biblioteca Pthreads

Pthread o POSIX *thread* como su nombre lo indica, es la biblioteca que reúne las condiciones necesarias para poder hacer uso y manejo de hilos, que puedan mantener la estabilidad y portabilidad del gestor de bases de datos PostgreSQL a la hora de paralelizar los procesos de ejecución de las consultas.

Esta biblioteca define una serie de estructuras y herramientas para lograr la sincronización, evitar corrupción de la memoria compartida y tratar cuestiones como el interbloqueo y la exclusión mutua entre los hilos. También posibilita que el propio planificador del sistema operativo se encargue de la planificación de los hilos, lo que evita que el bloqueo de uno estos, provoque el bloqueo de todo el proceso que ejecuta la gestión de la consulta.

2.4.1 Gestión de hilos

Para identificar los hilos en ejecución, *Pthread* define la estructura **`pthread_t`**, y para contener sus atributos **`pthread_attr_t`**. En dependencia de la modificación en sus atributos, un hilo tendrá un comportamiento determinado desde que inicia hasta que finaliza su ejecución.

La función **`pthread_create`** (**`pthread_t`** * *thread* , **`pthread_attr_t`** * *attr* , **`void`*** &*func* , **`void`** * *arg*), es la que permite crear un hilo para que comience su ejecución. Esta recibe cuatro parámetros:

1. *thread:* Es un puntero a una estructura **`pthread_t`,** que actúa como identificador del hilo, lo que nos permitirá realizar diferentes operaciones con él en cualquier momento de su ejecución.
2. *attr:* Es un puntero a una estructura **`pthread_attr_t`,** que debe inicializarse previamente con los atributos del hilo. Si este parámetro es NULL, el hilo tomará los valores por defecto.
3. *func*: Esta es la dirección a la función que debe ejecutar el hilo. Esta función debe recibir como parámetro un puntero genérico (**`void`** *) y devolver otro como resultado.
4. *arg:* Es un puntero al parámetro que se le pasará a la función que ejecuta el hilo. Puede ser NULL en caso que no se le pasen parámetros.

Para crear hilos dentro del proceso de PostgreSQL, se definieron algunos de los atributos siguientes:

- *contentionscope*: Precisa si el hilo será planificado por el sistema o por el mismo proceso. Se modifica su valor mediante la función **`pthread_attr_setscope`**. Por defecto lo planifica el proceso que lo crea, por tanto cambiamos su valor para que el sistema operativo sea el encargado de planificarlo y así se evita que las llamadas bloqueantes al sistema provoquen el bloqueo del proceso completo.
- *detachstate*: Define si otro hilo puede esperar a que este termine con su ejecución. Su valor es modificado por la función **`pthread_attr_setdetachstate`**. Se deja con su valor por defecto para hacer que el hilo principal del proceso espere a que los secundarios terminen su tarea.
- *policy:* Establece la política de planificación a aplicarse sobre el hilo a través de la función **`pthread_attr_setschedpolicy`**. Se deja con su valor por defecto para que el sistema decida qué política aplicar.
- *inheritsched:* Determina si los parámetros de planificación son heredados. Su valor se establece mediante la función **`pthread_attr_setinheritsched`**. Se deja el valor por defecto.
- *stackaddr:* Establece un puntero a la pila del hilo mediante la llamada a la función **`pthread_attr_setstackaddr`**. En este caso, como PostgreSQL tiene su propio sistema de alineación de memoria, este parámetro se cambia para que la memoria utilizada sea controlada dentro del proceso.
- *stacksize:* Fija el tamaño de la pila del hilo por medio de la función **`pthread_attr_setstacksize`**. El sistema por defecto le asigna 1MB. Se calcula un tamaño apropiado para evitar problemas con la gestión de memoria que realiza PostgreSQL.

En el caso de modificación de los atributos *stackaddr* y *stacksize*, se recomienda utilizar la función **`pthread_attr_setstack`** puesto que se pretende reemplazar las otras dos funciones debido a que modificar solamente el *stackaddr* puede hacer imposible la implementación en algunas arquitecturas.

2.4.2 Cerrojos

Las variables cerrojos, son una alternativa para dar solución al problema de la región crítica. *Pthread* define la estructura **`pthread_mutex_t`** que se comporta como una variable cerrojo. Para entrar y salir de las regiones críticas, se utilizan las funciones **`pthread_mutex_lock`** y **`pthread_mutex_unlock`** respectivamente. Ambas funciones reciben como parámetro la variable cerrojo correspondiente, y devuelven 0 en caso de completar con éxito el bloqueo, o cualquier otro valor en el caso contrario.

2.4.3 Semáforos

Los semáforos resuelven los mismos problemas que las variables de exclusión mutua, pero a diferencias de estas, el semáforo bloquea el hilo y no consume tiempo de ejecución en el CPU mientras espera para entrar en la región crítica. La biblioteca *Pthread*, define la estructura **`sem_t`**, para hacer uso de este tipo de variables. Para realizar las operaciones clásicas con semáforos, se utilizan las funciones **`sem_wait`** para poner a un hilo a esperar por una variable semáforo para continuar su ejecución y para desbloquear los hilos que esperan en el semáforo se utiliza la función **`sem_post`**. Ambas funciones reciben por parámetro un apuntador a la dirección del semáforo.

2.4.4 Monitores

Los monitores se utilizan para evitar el interbloqueo entre hilos que compitan por la misma región crítica y deban pasar en un orden condicionado. La biblioteca *Pthread* no tiene una estructura de finida para este tipo de variables, pero utiliza las condicionales definidas por la estructura **`pthread_cond_t`,** y las combina con las variables cerrojos para lograr el mismo comportamiento. Para hacer que un hilo espere por una condicional dentro de la región crítica, se utiliza la función **`pthread_cond_wait`** que recibe por parámetro un apuntador a una variable condición y otro a una variable cerrojo que bloquea la entrada a la región. Para hacer que el hilo que espera a que ocurra la condición continúe su ejecución, se envía una señal desde otro hilo utilizando la función **`pthread_cond_signal`,** la cual recibe como parámetro un puntero a la variable condición.

2.4.5 Barreras

Este es un mecanismo de sincronización para un grupo de hilos. Este mecanismo, generalmente, se utiliza en aplicaciones que están divididas en fases, donde existe la regla de que ningún proceso puede pasar a la siguiente fase, sin que el resto de los procesos haya terminado con la fase anterior.

En *Pthread* las barreras están definidas por la estructura **`pthread_barrier_t`**. Cuando se llega al punto de tener que esperar por el resto de los procesos se llama a la función **`pthread_barrier_wait`** que recibe como parámetro un apuntador a la dirección de la barrera.

2.5 Control de flujo para el procesamiento de consultas en el módulo ejecutor

En este epígrafe se describen las principales rutinas que se ejecutan durante el recorrido del árbol de descripción de consulta.

La clase *execMain.c* contiene las interfaces de rutinas de alto nivel del ejecutor.

Interfaces de rutinas:

- **`ExecutorStart()`**
- **`ExecutorRun()`**
- **`ExecutorFinish()`**
- **`ExecutorEnd()`**

Estos cuatro métodos son los procedimientos fundamentales de la interfaz externa para el ejecutor. En cada caso, el árbol descriptor de consulta se requiere como un argumento.

`ExecutorStart` debe ser llamado al comienzo de la ejecución de cualquier plan de consulta y **`ExecutorEnd`** siempre debe ser llamado al final de la ejecución de un plan (a menos que la ejecución haya sido abortada debido a un error).

`ExecutorRun` acepta los argumentos de la dirección y el recuento que especifican si el plan se va a ejecutar hacia delante, hacia atrás, y para cuántas tuplas. En algunos casos **`ExecutorRun`** puede ser llamado varias veces para procesar todas las tuplas de un plan.

`ExecutorFinish` debe ser llamada después de la llamada **`ExecutorRun`** final y antes de la llamada al método que finaliza el recorrido del árbol, **`ExecutorEnd`**. Esto se puede omitir solo en caso de EXPLAIN, lo que también se debe omitir **`ExecutorRun`**.

El fichero *execProcNode.c* contiene las funcionalidades encargadas de inicializar, obtener y limpiar las tuplas, que a su vez ejecutan las rutinas apropiadas para cada tipo de nodo dado. Si el nodo tiene hijos, entonces ejecutará las rutinas **`ExecInitNode`**, **`ExecProcNode`** o **`ExecEndNode`** en sus

subnodos para hacer el tratamiento adecuado y facilitar la sincronización cuando se procesan nodos nuevos.

Interfaces de rutinas:

- ❖ **`ExecInitNode()`** -- inicializa un nodo del plan
- ❖ **`ExecProcNode()`** -- obtiene una tupla mediante la ejecución del plan de nodo
- ❖ **`ExecEndNode()`** -- cierra un nodo del plan

Ejemplo:

Supongamos que queremos la edad del gerente del departamento de calzado y el número de empleados de ese departamento. Así que tenemos la consulta:

```
        Select DEPT.no_emps, EMP.age
where EMP.name = DEPT.mgr and DEPT.name = "shoe"
```

Supongamos que el planificador nos da el siguiente plan:

```
Nest Loop (DEPT.mgr = EMP.name)
           /          \
          /            \
 Seq Scan               Seq Scan
      DEPT             EMP
        (name = "shoe")
```

En el contexto de ejecución del ejemplo, **`ExecutorStart`** es llamado en primer lugar recibiendo como parámetro el árbol descriptor de la consulta, y este es el iniciador de la ejecución, que continúa con la llamada de **`InitPlan`** y este con la llamada **`ExecInitNode`** en la raíz del plan, en este ejemplo el nodo *`Nest Loop`*.

`ExecInitNode` detecta que está visitando un nodo *`Nest Loop`* y llama a **`ExecInitNestLoop`**. Eventualmente, se realizan las llamadas del método **`ExecInitNode`** en los árboles izquierdo y derecho que contienen subplanes y, así sucesivamente, hasta que todo el plan se ha inicializado. El resultado devuelto por el método **`ExecInitNode`** es un árbol del plan de estado construido con la misma estructura que el árbol del plan subyacente.

Luego de inicializados los árboles plan y plan de estado, se ejecuta la rutina **`ExecutorRun`**, que inicia la ejecución de cada uno de los nodos del plan de estado, lo cual se realiza a través de la llamada al método **`ExecutePlan`** que llama a **`ExecProcNode`** varias veces en el nodo superior del árbol del plan

de estado. En cada ejecución del método **ExecProcNode**, se va a llamar al método correspondiente por el tipo de nodo que se visite, en el caso del ejemplo sería **ExecNestLoop**, pues su función consiste en llamar a **ExecProcNode** en cada uno de sus subplanes.

Cada uno de estos subplanes, es una secuencia de exploración donde **ExecSeqScan** es el método llamado. Las ranuras[7] devueltas por **ExecSeqScan**, pueden contener tuplas portadoras de los atributos que **ExecNestLoop** utiliza para formar las tuplas que deben ser devueltas al cliente al fin de la ejecución. Esta ejecución finaliza, cuando **ExecSeqScan** retorna las tuplas y **ExecNestLoop** une las tuplas retornadas de sus 2 extremos o subplanes.

Una vez ejecutados cada uno de los nodos contenidos en el árbol descriptor de la consulta, y antes de hacer la llamada final al **ExecutorRun**, se debe ejecutar la rutina **ExecutorFinish** que es la encargada de la llamada de las funciones **ExecPostprocessPlan** y **AfterTriggerEndQuery** programadas para ejecutar todos los nodos de alto nivel *ModifyTable* sin terminar, que contienen la información de tupla necesaria para el retorno al cliente.

Por último, la rutina encargada de finalizar la ejecución y apagar el ejecutor es **ExecutorEnd**. El proceso continúa con la llamada al método **ExecEndNode**, que llama a **ExecEndNestLoop**, responsable de la llamada del método **ExecEndNode** en cada uno de sus subplanes, lo cual resulta, en el caso específico del ejemplo, en la llamada a **ExecEndSeqScan**.

Anteriormente, se describe el proceso desencadenado al realizarse una llamada el ejecutor de PostgreSQL. **ExecutorStart**, **ExecutorRun**, **ExecutorFinish** y **ExecutorEnd** son las interfaces de rutinas principales del módulo, encargadas de las llamadas a **ExecInitNode**, **ExecProcNode** y **ExecEndNode** métodos base, encargados de distribuir el trabajo a ejecutarse por las rutinas de apoyo apropiadas para cada tipo de nodo y sus subplanes, visitados durante el recorrido realizado al árbol descriptor de la consulta.

[7] **ranura (slot): tipo de dato de retorno para los métodos que devuelven una tupla durante el recorrido o ejecución de un árbol descriptor de consulta en PostgreSQL.**

Este es un boceto de control de flujo para el procesamiento de consultas completo:

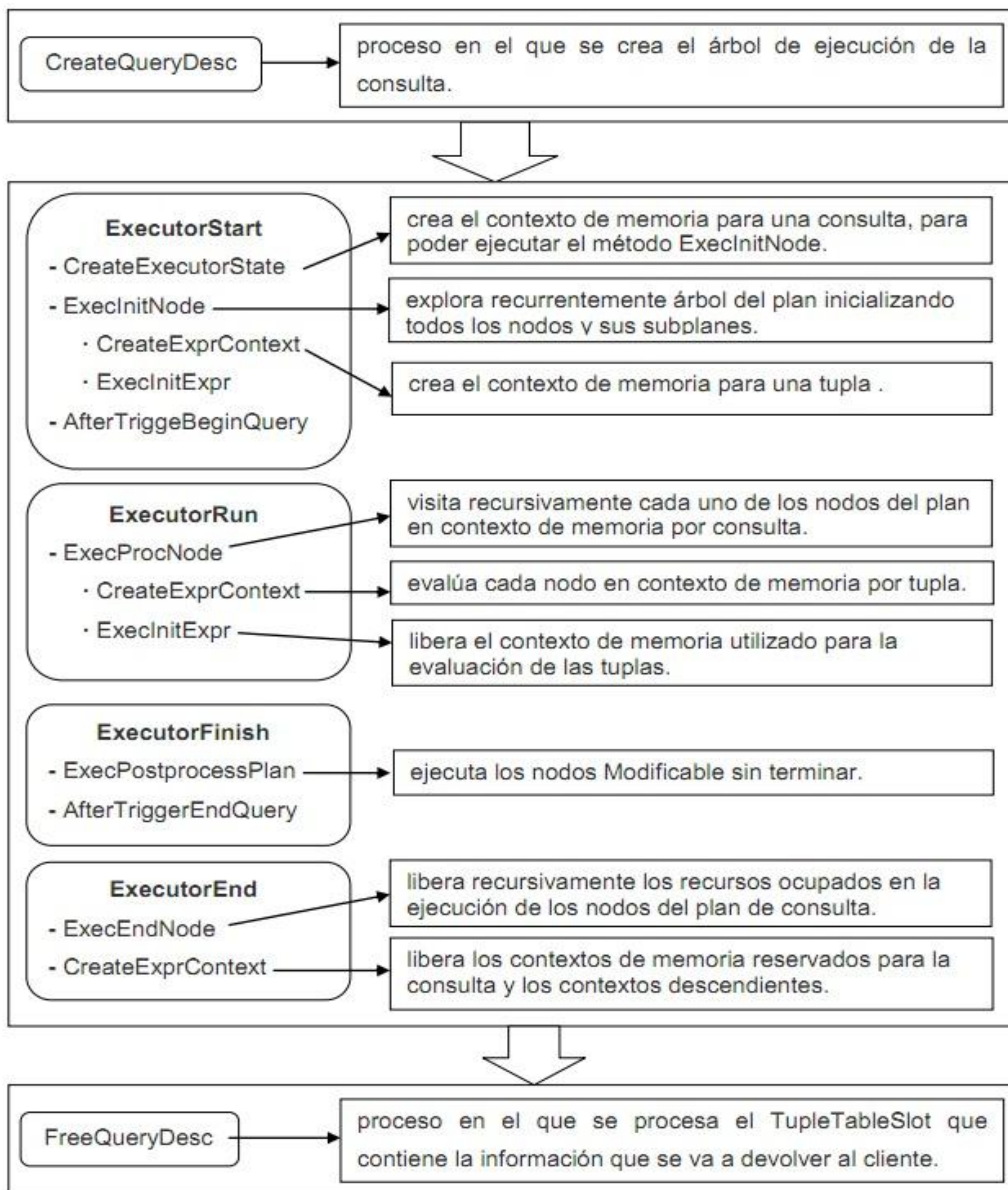

Figura 2.1- Control de flujo para el procesamiento del plan de consulta.

2.6 Fundamentos de la solución propuesta

Dentro del módulo ejecutor del servidor PostgreSQL, la función **`standard_ExecutorRun`** es la que comienza el recorrido del árbol de nodos del plan de una consulta. Para la recopilación de las tuplas, se hace una llamada al método **`ExecutePlan`** con los parámetros necesarios para llevar a cabo la ejecución del plan y obtener todas las tuplas. Este proceso es cíclico y consta de varios pasos que dividiremos en dos procesos **A** y **B**:

A- Se obtiene una tupla y en caso de ser la última, se sale del ciclo de ejecución.

B- Se filtra una tupla en caso de que la misma al ser obtenida contenga información distinta a la contenida en el descriptor de tupla, por lo que se recopilan los datos necesarios en dependencia del destino correspondiente, se incrementa el contador de tuplas y se chequea si ha llegado a la cantidad de tuplas definida en la consulta.

En el proceso **A** es donde se comienza a recorrer el árbol del plan de ejecución a un nivel inferior. En este nivel los nodos que se ejecutan, son divididos en dos grupos principales, nodos físicos y nodos virtuales.

Los nodos físicos, como el *`T_ScanState`*, realizan operaciones directamente sobre el clúster de datos donde se encuentran las tuplas físicas de las tablas almacenadas. Al no haber forma de acceder directamente a una tupla específica que no sea la primera o la última tupla, la recopilación de tuplas debe hacerse de forma secuencial obligatoriamente.

Los nodos virtuales, como el *`T_HashJoinState`*, tienen la mayor carga de trabajo sobre tablas almacenadas en memoria, pero transitan por diversos estados en su ejecución lo que vuelve extremadamente complicado realizar una ejecución paralela dentro de estos nodos sin cambiar su estructura.

En el proceso **B**, el filtrado solo se ejecuta en algunos casos, ya que no ocurre para la totalidad de las consultas. La tupla se pasa a una función que varía en dependencia de a dónde haya que enviar el resultado de la consulta, generalmente va escribiendo en un *buffer* los datos necesarios de la tupla obtenida. Y por último, con un contador se controla la cantidad de tuplas a procesar.

Dado lo expuesto anteriormente, se puede concluir que no es factible trabajar con hilos en el nivel inferior sin tener que realizar grandes cambios en la estructura del plan de ejecución de las consultas; tampoco se pueden realizar ejecuciones en paralelo del proceso **A**, ni se pueden escribir paralelamente los resultados de las tuplas que son recopiladas por el proceso **B**.

Se puede apreciar que **A** puede ejecutarse para producir una tupla, al mismo tiempo que **B** se ejecuta sobre otra producida anteriormente. El modelo Productor-Consumidor puede ser apropiado en este

caso. También un determinado proceso **C**, formado por la unión de los dos procesos anteriores, podría ejecutarse de forma paralela, donde **A** y **B** serían regiones críticas. Se pueden crear hilos y sincronizarlos para que se ejecutasen en paralelo sin que haya dos hilos trabajando sobre la misma región crítica.

Para ver si estas soluciones son factibles, partimos de que una tupla debe pasar por el proceso **A** donde se produce con un tiempo ejecución T_A y luego por el proceso **B** donde se almacena o consume con un tiempo T_B, el proceso completo demoraría un tiempo T_C igual a la suma de los tiempos T_A y T_B. Entonces el tiempo de ejecución secuencial del proceso **C** para procesar todas las tuplas, se obtendría por la fórmula aritmética: $T_C = (T_A + T_B) * n$, donde **n** representa la cantidad de tuplas a procesar.

Si se ejecutase el proceso **C** de forma paralela, tardaría un tiempo de ejecución que sería igual al tiempo secuencial T_C, dividido entre los **h** hilos que procesan las **n** tuplas paralelamente. Por tanto podemos decir que el tiempo de ejecución del proceso **C** si se ejecutara de forma paralela T_C', sería igual a la suma de los tiempos de ejecución de los procesos **A** y **B** multiplicados por la cantidad de tuplas a procesar, dividido entre la cantidad de hilos que vayan a ejecutar el proceso **C** paralelamente: $T_C' = (T_A + T_B) * n / h$.

Como se aclaró anteriormente, solo puede haber un hilo a la vez creando tuplas en el proceso **A**, por lo que no habrá ninguno otro hilo trabajando cuando se inicie la ejecución del proceso **C**, de igual forma no pueden haber dos hilos escribiendo el resultado, por tanto cuando se estén guardando los datos de la última tupla en el proceso **B**, tampoco habrá ninguna procesándose en los hilos restantes al terminar la ejecución del proceso **C**. Dado esto, se puede establecer que la suma del tiempo de ejecución que demora en procesarse la primera y la última tupla, equivale al procesamiento de una tupla en una ejecución paralela. La fórmula para el cálculo del tiempo de ejecución paralela del proceso **C** quedaría planteada como: $T_C' = (T_A + T_B)*(n - 1) / h$.

La diferencia entre este enfoque y un modelo Productor-Consumidor, consiste en que **A** y **B** dejarían de ser regiones críticas del proceso **C** y pasarían a ejecutarse en distintos hilos. Pero el tiempo de ejecución en ambos casos es el mismo.

Si se le asignaran valores hipotéticos a las variables de la fórmula definida para el cálculo de los tiempos de ejecución, se puede tener una idea de qué tan factible es la solución paralela con respecto al algoritmo secuencial.

Supongamos que $T_A = 0.5$ ut[8] y $T_B = 0.4$ ut. En una consulta que retorne N = 10000 tuplas.

[8] **ut: Unidades de tiempo**

	Fórmulas:	Resultados en Ut:
Ejecución Secuencial	$\mathbf{T_C} = (\mathbf{T_A} + tB)*N.$	$\mathbf{T_C} = 9000$ ut
Ejecución Paralela	$\mathbf{T_C}' = (\mathbf{TA} + tB)*(N - 1) / 2$	$\mathbf{T_C}' = 4499,55$ ut

Tabla 2.1- Resultados aritméticos del cálculo de los tiempos de ejecución de los algoritmos secuencial y paralelo.

Como podemos apreciar en la tabla anterior, el tiempo de ejecución que tarda el *backend*[9] en procesar las tuplas se disminuye a la mitad aproximadamente. Al llevar el algoritmo a la práctica, esta investigación científica promete resultados satisfactorios.

Debido a la extrema complejidad que representa el estudio y modificación del código fuente del servidor de bases de datos PostgreSQL, se centró el estudio en un tipo específico de consulta compleja de tipo SELECT, que realiza JOIN entre varias tablas, dado que es una de las consultas que más cómputo realiza por lo que se diseñó e implementó el algoritmo paralelo para su ejecución. Para la descripción de la solución se utilizó una de las bases de datos (*pagila*) y una de las consultas oficiales más costosas utilizadas para pruebas por la comunidad oficial de PostgreSQL, la misma que devuelve un total aproximado de 5462 tuplas de la base de datos *pagila*.

Supongamos que queremos obtener todos los apellidos de los actores que han trabajado en alguna película anterior a la fecha actual. Ejecutamos la sentencia sql:

```
SELECT actor.last_name FROM actor LEFT JOIN film_actor USING (actor_id)
WHERE Actor.last_update < current_date;
```

El planificador crea el siguiente plan:

```
                 HashJoinState(actor left join film_actor)
                          /               \
                        /                   \
              ScanState                      HashState
     (last_update < current_date)          (Tabla Hash)
```

El nodo `HashJoinState` es una máquina de estados, donde cada estado representa una etapa en la devolución de cada tupla. En la primera ejecución se ejecuta el hijo derecho (`HashState`); el cual devuelve la tabla hash, o la carga de la caché si ya ha sido creada con anterioridad, conteniendo los valores del `Left Join`. Luego verifica si debe hacer una ejecución del hijo izquierdo y retorna la primera tupla en caso de que tenga que hacerlo. En la próxima ejecución del `HashJoinState`, solo se ejecuta el hijo izquierdo en busca de una nueva tupla que cumpla la condición para devolverla, y así

[9] **backend: proceso que representa una conexión cliente/servidor para la gestión de la consulta, independiente e invisible para el usuario.**

repite hasta que llega a la última tupla, entonces almacena la tabla hash en la cache, por si hace falta realizar otra búsqueda en la misma tabla hash y retorna la tupla nula.

Tomando como solución solo el proceso de ejecución de consultas complejas de tipo SELECT con JOIN, se propone la modificación del módulo ejecutor del servidor PostgreSQL. La solución propuesta responde a los siguientes diagramas:

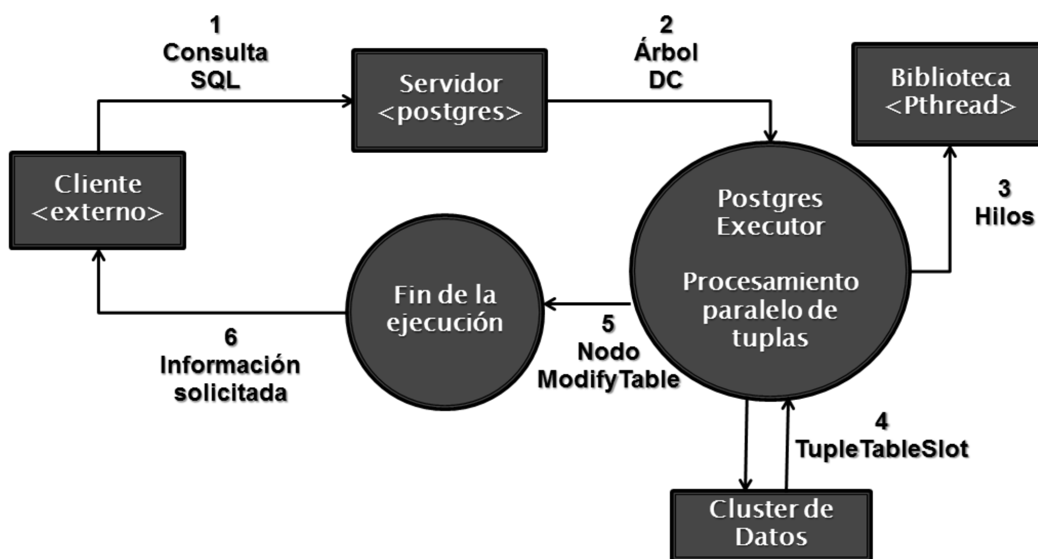

Figura 2.2-Diagrama de Flujo para la ejecución paralela de una consulta.

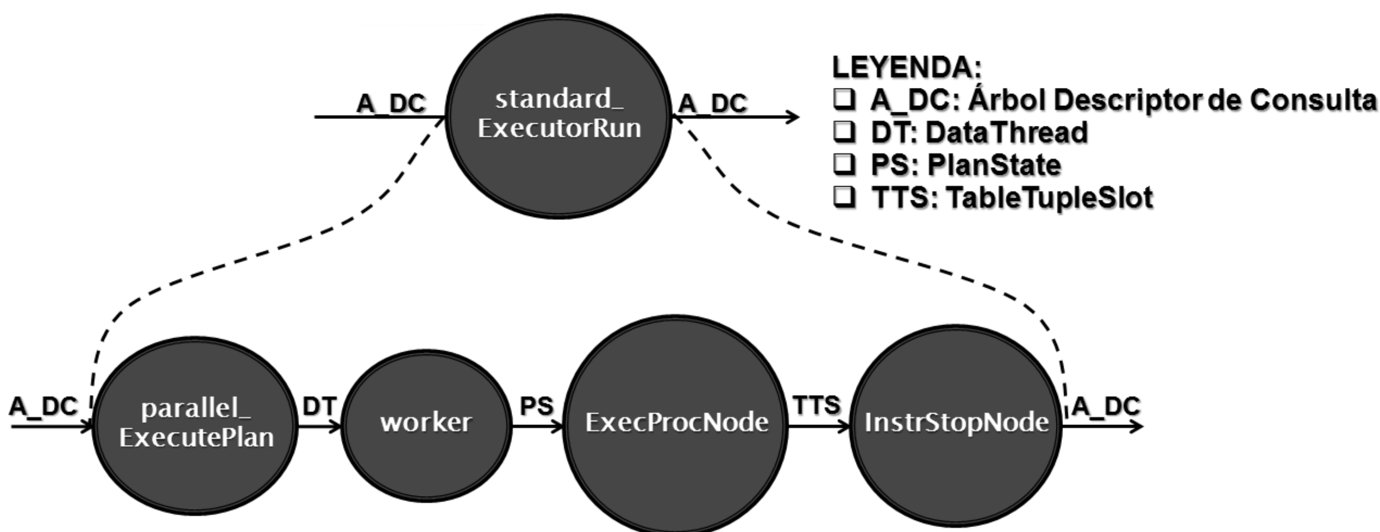

Figura 2.3-Diagrama de Flujo de Datos para el procesamiento paralelo de tuplas durante la ejecución de una consulta.

2.7 Conclusiones del capítulo 2

En este capítulo, se definieron las principales herramientas y tecnologías utilizadas durante la investigación para el desarrollo del algoritmo paralelo propuesto para el procesamiento de tuplas. Se realiza una breve descripción de algunas de las funciones que permiten gestionar los hilos, semáforos, barreras, entre otras funcionalidades básicas de la librería de C/C++ *Pthread*. Se mencionan algunas de las características y se describen las principales rutinas ejecutadas durante el recorrido que realiza el módulo ejecutor de PostgreSQL sobre el árbol de descripción de consulta. Se describe en detalles la solución propuesta para paralelizar el procesamiento de las tuplas contenedoras de la información solicitada por el cliente.

CAPÍTULO 3: IMPLEMENTACIÓN, PRUEBAS Y RESULTADOS

3.1 Introducción

En este capítulo, se describen las pruebas diseñadas para la validación de los resultados de la investigación, los fundamentos y los principales componentes de la solución propuesta así como los mecanismos de implementación utilizados. Se mencionarán los resultados de la investigación.

3.2 Implementación de la solución propuesta

Para llegar a la solución se implementaron dos variantes del modelo de algoritmo paralelo Productor-Consumidor, las cuales se explicarán posteriormente. Primero se debe aclarar que, durante el proceso de ejecución de una consulta, en el *backend* se chequean algunos parámetros para asegurar que no existen errores en tiempo de ejecución.

Uno de estos parámetros, chequeado muy a menudo, es la profundidad del *stack*[10] del proceso. Este chequeo se realiza mediante la llamada a la función **`check_stack_depth`**, para evitar el desbordamiento de la memoria del proceso. Para realizar esta comprobación, se crea una variable de tipo *char,* que mide la distancia que hay desde el principio de la memoria reservada para el proceso, hasta la dirección donde se crea esta variable. Si la cantidad de memoria dentro de esta distancia es mayor que el tamaño de la memoria reservada para el *stack* al inicio del programa, el *backend* interrumpe su ejecución mostrando el error "*stack depth limit exceded*". Como la arquitectura del *backend* no está pensada para trabajar con hilos, la aritmética utilizada en este caso es inexacta, debido a que cada hilo contiene su propio *stack*, por tanto se inhabilitó este control para poder llevar a cabo la ejecución del algoritmo paralelo.

Dentro del módulo ejecutor hay dos funciones fundamentales, las cuales permiten crear y almacenar las tuplas de una consulta. Estas funciones son ejecutadas dentro de un ciclo infinito, que termina cuando se encuentra la última tupla de la consulta. A continuación se da una breve descripción de ambas:

- ✓ *ExecProcNode*: Recibe como parámetro el árbol del `PlanState`, y ejecuta una función en correspondencia con el tipo de nodo del árbol que se vaya a procesar. Retorna un puntero a una estructura del tipo `TupleTableSlot`. Esta es la función que el *Executor* llama cíclicamente hasta obtener la última tupla a producir en la consulta.
- ✓ *receiveSlot*: Es un puntero dirigido a la función que debe ejecutarse para guardar los datos que devuelve la consulta. Está contenido dentro de una estructura de tipo `DestReceiver`. Este

[10] **stack: pila utilizada por los procesos para almacenamiento en memoria de los datos durante su ejecución.**

puntero está dirigido a una función distinta en dependencia del proceso a realizar con los datos de la tabla. Recibe dos parámetros, un puntero a una estructura *TupleTableSlot* y el segundo es la propia estructura *DestReceiver* que contiene al *receiveSlot*.

La estructura *TupleTableSlot* está formada por una serie de variables que posibilitan realizar las operaciones necesarias en la ejecución de la consulta. Una de estas variables contiene la información que forma parte del resultado de la consulta.

Se debe tener en cuenta que la llamada de la función **ExecProcNode** retorna un puntero a la dirección del *TupleTableSlot* de la tupla obtenida. Al no tratarse de una copia de los datos del *TupleTableSlot*, la acción de obtener y procesar tuplas no podría ejecutarse al mismo tiempo sobre tuplas diferentes sin que se alterase la información contenida en ellas, por lo que se implementó la función **copySlot** para crear una copia de los datos que forman parte del resultado a devolver por la consulta. Esta función recibe como parámetros dos estructuras *TupleTableSlot*; una es el destino y la otra la fuente de la información a copiar.

Las funciones **ExecProcNode** y **receiveSlot** hacen uso de una serie de estructuras que posibilitan llevar a cabo el procesamiento de las tuplas. En la siguiente tabla, se presenta una breve descripción de estas estructuras.

Tipo de estructura	Identificador	Descripción
EState	estate	Es un puntero a una estructura que almacena el estado de ejecución en el módulo del ejecutor.
PlanState	planestate	Es una interfaz para manejar las estructuras de todos los nodos del plan de ejecución de las consultas de PostgreSQL.
CmdType	operation	Enumerador para identificar el tipo de operación que realiza la consulta.
bool	sendTuples	Variable lógica para saber si la consulta devuelve algún dato.
long	numberTuples	Número entero largo para contar las tuplas a producir. Toma el valor 0 para retornar todas las tuplas encontradas.
ScanDirection	direction	Determina la dirección de escaneo de las tuplas.

`DestReceiver`	dest	Es una estructura que contiene los punteros a las funciones que procesan los datos dentro del *`TupleTableSlot`* para enviarlos a los diferentes destinos.

Tabla 3.1- Descripción de las estructuras más utilizadas en el *backend* durante el procesamiento de las tuplas.

Para la ejecución de los algoritmos implementados, se deben compartir estas estructuras entre cada uno de los hilos creados, debido a que tener una copia en cada hilo, provocaría duplicación de todas las tuplas de la consulta. Para que los hilos compartan estas estructuras, se agruparon dentro de una estructura creada con el identificador DataThread.

Para gestionar los hilos, y atribuirle las características necesarias para el proceso, se utilizaron las siguientes estructuras:

Tipo de estructura	**Identificador**	**Descripción**
`pthread_t`	threads	Es un arreglo de la estructura de pthread que sirve como identificador para los hilos.
`pthread_attr_t`	attr	Es un arreglo de la estructura pthread que sirve para manejar los parámetros de creación de los hilos.
`void *`	stack	Es un puntero a la dirección del espacio en memoria que reservamos para la pila de los hilos.

Tabla 3.2- Descripción de las estructuras utilizadas en la gestión de los hilos.

Para llevar a cabo la implementación de la primera variante, se creó un algoritmo que compartiera la carga de trabajo entre los hilos. En este caso, cada hilo debe contener su propio *buffer* donde pone las tuplas temporalmente para luego procesarlas. Los hilos deben estar sincronizados para que cuando uno termine de producir tuplas, el otro comience a llenar su *buffer*. En la siguiente tabla mostramos las herramientas de *pthread* utilizadas para la sincronización de los hilos en esta variante.

Tipo de estructura	**Identificador**	**Descripción**
`pthread_mutex_t`	prod	Estructura de *Pthread* usada para lograr la exclusión mutua entre los hilos en la primera región crítica.
`pthread_mutex_t`	cons	Estructura de *Pthread* usada para lograr la exclusión mutua entre los hilos en la segunda región crítica.

`sem_t`	start	Semáforo de *Pthread* utilizado para iniciar el trabajo de los hilos.
`char`	finish	Variable para comunicarle a un hilo que el otro procesó la última tupla de la consulta.

Tabla 3.3- Descripción de las estructuras utilizadas en la sincronización de los hilos de la primera variante.

Para organizar el entorno de ejecución de los hilos, se implementó la función **`parallel_ExecutePlan`**, y cada hilo ejecuta el algoritmo **`worker`** para comenzar con el trabajo asignado. A continuación se representan ambos pseudocódigos:

a) *parallel_ExecutePlan*	**b)** *worker*
1: **inicializar el** *DataThread* 2: **iniciar hilos** 3: **para** i=0 **hasta** cantidad de hilos **hacer** 4: **subir semáforo** *start* 5: **fin para** 6: **para** i=0 **hasta** número de hilos **hacer** 7: esperar que el $hilo_{(i)}$ termine de trabajar 8: **fin para**	**variables**: *fin ←false* 1: **bajar semáforo** *start* 2: **mientras** *true* **hacer** 3: **bloquear** *regionCríticaA* 4: **si** *fin* **enconces** 5: *salir* 6: **fin si** 7: **para** *i = 0* **hasta** *size(buffer)* **hacer** 8: S ← *obtenerTupla()* 9: **si** *vacío*(S) **entonces** 10: *fin ← true* 11: *salir* 12: **fin si** 13: **copiar** S **en** *buffer*[i] 14: **fin para** 15: **desbloquear** *regionCríticaA* 16: **bloquear** *regionCríticaB* 17: **para** *i = 0* **hasta** *size(buffer)* **hacer** 18: **si** *vacío*(*buffer*[i]) **entonces** 19: *salir* 20: **fin si** 21: *enviar*(*buffer*[i]) 22: **fin para** 23: **desbloquear** *regionCríticaB* 24: **si** *fin* **enconces** 25: *salir* 26: **fin si** 27: **fin mientras**

Tabla 3.4- Pseudocódigo de los algoritmos (a) parallel_ExcutePlan y (b) worker de la primera variante.

En la segunda propuesta, se implementó una variante del modelo productor consumidor con dos *buffer*. Para la sincronización de los hilos se utilizaron algunas estructuras de *pthread* de las cuales se presenta una breve descripción en la siguiente tabla.

Tipo de estructura	Identificador	Descripción
`sem_t`	cons	Semáforo usado para que el hilo consumidor espere a que se llene un *buffer*.
`sem_t`	sw	Semáforo usado para que el hilo productor no pueda cambiar de *buffer* a procesar sin que el hilo consumidor haya terminado con él. Inicia en 1 para cuando se llene el primer *buffer*.
`TupleTableSlot`	buffer	Arreglo de *`TupleTableSlot`* que conforma los dos *buffer*s de esta variante.

Tabla 3.5- Descripción de las estructuras utilizadas en la sincronización de los hilos de la segunda variante.

Cuando el hilo productor termina de llenar un *buffer* comienza a producir en el otro y avisa al consumidor qué *buffer* está listo para procesarse. El consumidor comienza a procesar las tuplas almacenadas en el *buffer* lleno. Para esta variante se implementó la función **`parallel_ExecutePlan`**, la cual cumple la misma función que en la propuesta anterior, aunque varía un poco su algoritmo. También se implementaron dos algoritmos, **`workerP`**: encargado de producir las tuplas y **`workerC`**: encargado de procesarlas.

a) *parallel_ExecutePlan*	**b)** *workerP*
1: **inicializar el** *DataThread* 2: **iniciar el** *buffer* 3: **iniciar semáforos** *cons* ← 0, *sw* ←1 4: **iniciar** atributos de los hilos 5: **iniciar** hilo productor 6: **iniciar** hilo consumidor 7: **esperar** por el hilo productor 8: **esperar** por el hilo consumidor	1: **mientras** *true* **hacer** 2: **para** *i = 0* **hasta** *size(buffer)* **hacer** 3: S ← *obtenerTupla()* 4: **si** *vacío*(S) **entonces** 5: *buffer*[i].tupla_vacía ← *true* 6: *salir* 7: **fin si** 8: **copiar** S **en** *buffer*[i] 9: **fin para** 10: **bajar semáforo** *sw* 11: **seleccionar** *buffer* para procesar 12: **seleccionar** *buffer* para llenar 13: **subir semáforo** *cons* 14: **si** *vacío*(S) **entonces** 15: *salir* 16: **fin si** 17: **fin mientras**
c) *workerC* **variables**: *procesar* ← *true* *nTuplas* ← 0 1: **mientras** *procesar* **hacer** 2: **bajar semáforo** *cons* 3: **para** *i = 0* **hasta** *size(buffer)* **hacer** 4: **si** *vacío*(*buffer*[i]) **entonces** 5: procesar ← *false* 6: *salir* 7: **fin si** 8: *enviar*(*buffer*[i]) 9: *nTuplas* ← *nTuplas* + 1 10: **fin para** 11: **subir semáforo** *sw* 12: **fin mientras**	

Tabla 3.6- Pseudocódigo de los algoritmos (a) parallel_ExcutePlan, (b) workerP y (c) workerC de la segunda variante.

El proceso de obtención de una tupla, es un tanto más complejo que el resto de las operaciones realizadas sobre la tupla para devolver la información resultante de la consulta a su destino. La primera variante se plantea balancear la carga de trabajo entre los hilos, y hacer que la sincronización entre los hilos no provoque un cambio constante en el estado de ejecución del hilo. La desventaja de esta

variante se encuentra en que PostgreSQL implementa sus propios métodos de manejo de la memoria. Cada consulta tiene un contexto de memoria, y para el proceso de obtención de las tuplas se crea otro contexto el cual es reiniciado luego de producir una tupla. Esto hace que al producir tuplas utilizando múltiples hilos existan inconsistencias, dado que un cambio del contexto de memoria en varios hilos tiene un comportamiento indefinido. La segunda variante, es la solución concluyente para este tipo de consultas, dado que los cambios de contexto solo ocurren dentro del hilo del productor, por tanto un solo hilo trabaja en el manejo de memoria en el contexto de la consulta. Por otra parte, en la ejecución de los diferentes tipos de consultas, se presentan algunas inconsistencias en la reserva y liberación de memoria cuando se utilizan hilos, debido a que cada consulta trata la memoria de una manera muy específica y no existen regularidades en el tratamiento de la memoria entre las diferentes ejecuciones.

3.3 Diseño de los Casos de Prueba

Para medir la eficiencia del algoritmo paralelo desarrollado para el procesamiento de las tuplas se diseñaron 2 pruebas fundamentales, una para medir el tiempo de ejecución, la ganancia de velocidad y la eficiencia del modelo del algoritmo productor-consumidor implementado y otra para medir el rendimiento del *backend* al ejecutar las consultas seleccionadas para las pruebas utilizando los algoritmos secuencial y paralelo. Para la ejecución de las pruebas se utilizó una base de datos de alrededor de 17 millones de tuplas de datos biológicos, garantizando así un gran procesamiento de información de tupla durante la ejecución de las pruebas diseñadas.

3.3.1 Pruebas del algoritmo paralelo

Para medir el tiempo de ejecución del algoritmo ($\mathbf{T_E}$) se declararon dos variables para medir los tiempos inicial ($\mathbf{t_I}$) y final ($\mathbf{t_F}$) de la ejecución secuencial y paralela, y de esta manera poder determinar a través de la resta del tiempo final menos el inicial el valor real del tiempo aritmético ($\mathbf{T_A}$) que demora la ejecución. En el caso específico de PostgreSQL, la fórmula definida para el cálculo del tiempo de ejecución sería $T_E \approx T_A$ debido a que no existe comunicación ni dependencia con ningún otro proceso y esto permite despreciar el $\mathbf{T_C}$ que aparece propuesto en la fórmula 1.2 planteada en el epígrafe 1.3.2.

Para la medición de la ganancia en velocidad ($\mathbf{S_P}$) se procedió al cálculo del tiempo de ejecución ($\mathbf{T_E}$) para el algoritmo secuencial ($\mathbf{T_S}$) y para el paralelo ($\mathbf{T_P}$) y se calculó utilizando la fórmula 1.3 planteada en el epígrafe 1.3.2.

Para medir la eficiencia (**E**) del algoritmo paralelo se procedió utilizando la fórmula 3.4 planteada en el epígrafe 1.3.2. El valor de ganancia de velocidad dividido entre dos, que es la cantidad de

procesadores que posee el equipo utilizado para la realización de las pruebas, sería el valor de eficiencia del algoritmo paralelo implementado para la solución propuesta.

3.3.2 Pruebas de rendimiento para el servidor PostgreSQL

El experimento diseñado para la evaluación de rendimiento del servidor PostgreSQL consiste en la ejecución reiterada de la consulta de prueba utilizando la versión 9.1.3 del servidor que realiza el procesamiento de tuplas secuencialmente y posteriormente utilizando la misma versión pero procesando las tuplas en forma paralela. Las ejecuciones se desarrollaron sobre una base de datos biológicos contenedora de 17 millones de tuplas aproximadamente. De los resultados de las ejecuciones se despreciaron el mayor y el menor tiempo, calculándose la media entre los valores restantes. Por tanto: **T** = **∑t(i)** / **n-2**, donde **T** representa el valor total de tiempo en segundos que demoró la ejecución de la consulta, **t(i)** es el tiempo de una ejecución de la consulta, **n** el total de veces que se ejecutó la consulta de prueba e **i** varía en el rango de 2…n-1, considerando, sin pérdida de generalidad, los tiempos **t(i)** ordenados.

Consulta propuesta para las pruebas de rendimiento:

- ✓ SELECT id, entry_id, id_seqxml, type, dbref.source from dbref left join seqxml using (id_seqxml);

3.4 Resultados

Para la ejecución de las pruebas, se destinó un ordenador de hardware similar al descrito en el epígrafe 2.3, utilizando el sistema operativo Debian Wheezy con la versión 3.2 del kernel de Linux. Las pruebas se realizaron consecutivamente, en cada servidor como se describió anteriormente en los casos de prueba y cuyos resultados se muestran a continuación:

# Iter.	Tuplas Proc.	T. Ejec. Sec. $T_S(i)$	T. Ejec. Par. $T_P(i)$
1	4 213 868	13 s	18 s
2	4 213 868	14 s	17 s
3	4 213 868	12 s	18 s
4	4 213 868	11 s	17 s
5	4 213 868	12 s	17 s
6	4 213 868	12 s	17 s
7	4 213 868	13 s	17 s
8	4 213 868	11 s	16 s

9	4 213 868	12 s	18 s
10	4 213 868	13 s	16 s

Tabla 3.7-Resultados de las 10 iteraciones de prueba para un total de 4 213 868 tuplas a procesar.

T. Sec. Total	T. Par. Total	Ganancia de Velocidad	Eficiencia
12.25 s	17.125 s	0.715	0.357

Tabla 3.8-Resumen de resultados de la prueba de rendimiento.

Como se puede apreciar en las tablas 3.7 y 3.8, después de realizar 10 iteraciones de pruebas procesando 4 213 868 de tuplas, el cálculo de la eficiencia del algoritmo paralelo respecto al secuencial reportó un valor por debajo de 0.5, indicando que para procesar esta cantidad de tuplas el algoritmo es poco eficiente.

Debido a los resultados deficientes de la prueba anterior se decide aumentar la cantidad de tuplas a procesar en los siguientes casos de prueba:

# Iter.	Tuplas Proc.	T. Ejec. Sec. $T_S(i)$	T. Ejec. Par. $T_P(i)$
1	8 206 816	45 s	35 s
2	8 206 816	26 s	38 s
3	8 206 816	47 s	50 s
4	8 206 816	50 s	42 s
5	8 206 816	42 s	39 s
6	8 206 816	32 s	35 s
7	8 206 816	36 s	38 s
8	8 206 816	42 s	33 s
9	8 206 816	45 s	33 s
10	8 206 816	37 s	40 s

Tabla 3.9-Resultados de las 10 iteraciones de prueba para un total de 8 206 816 tuplas a procesar.

T. Sec. Total	T. Par. Total	Ganancia de Velocidad	Eficiencia
40.75 s	37.5 s	1.086	0.543

Tabla 3.10-Resumen de resultados de la prueba de rendimiento.

# Iter.	Tuplas Proc.	T. Ejec. Sec. $T_S(i)$	T. Ejec. Par. $T_P(i)$
1	11 297 925	205 s	88 s
2	11 297 925	347 s	62 s
3	11 297 925	114 s	87 s
4	11 297 925	198 s	187 s
5	11 297 925	800 s	50 s
6	11 297 925	616 s	58 s
7	11 297 925	188 s	209 s
8	11 297 925	86 s	57 s
9	11 297 925	39 s	335 s
10	11 297 925	151 s	493 s

Tabla 3.11-Resultados de las 10 iteraciones de prueba para un total de 11 297 925 tuplas a procesar.

T. Sec. Total	T. Par. Total	Ganancia de Velocidad	Eficiencia
238.125 s	135.375 s	1.759	0.879

Tabla 3.12-Resumen de resultados de la prueba de rendimiento.

Como se puede apreciar en las tablas 3.9 y 3.10, ha ocurrido un cambio positivo en los resultados del experimento, porque al procesarse 8 206 816 tuplas, ya se observa un incremento de la eficiencia del algoritmo paralelo, además de una ganancia de velocidad de 1.068 (de un máximo teórico posible de 2).

Los resultados apreciados en las tablas 3.11 y 3.12, son satisfactorios porque después de haberse procesado 11 297 925 tuplas, en los cálculos de eficiencia y ganancia de velocidad se obtuvieron valores de 0.879 y 1.759 lo que demuestra que la solución propuesta es factible.

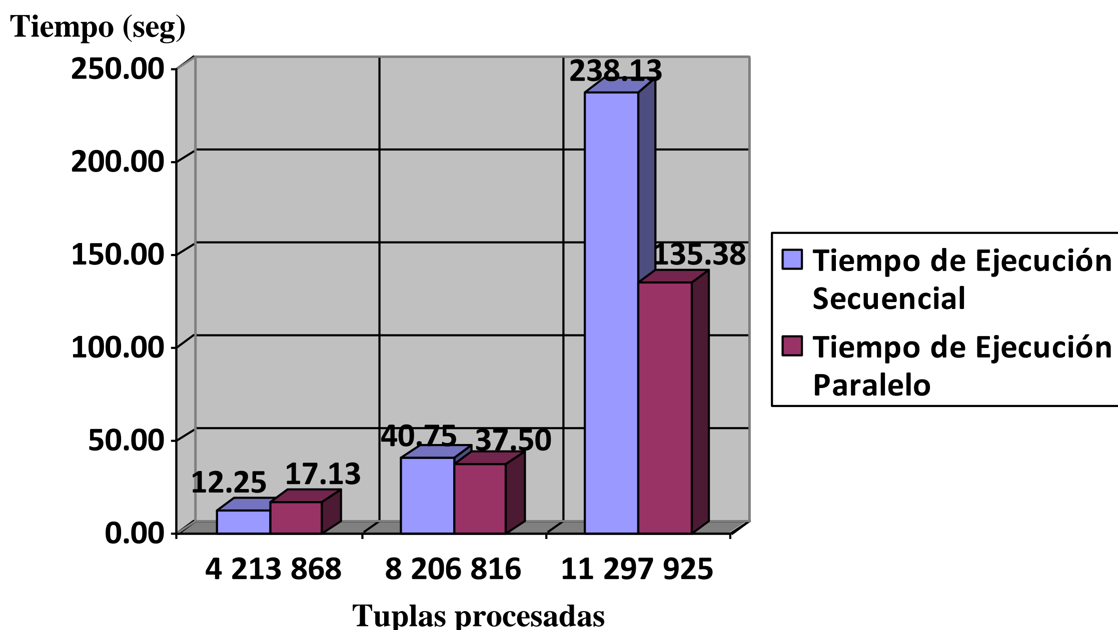

Figura 3.1-Resumen de resultados de las prueba de tiempo de ejecución.

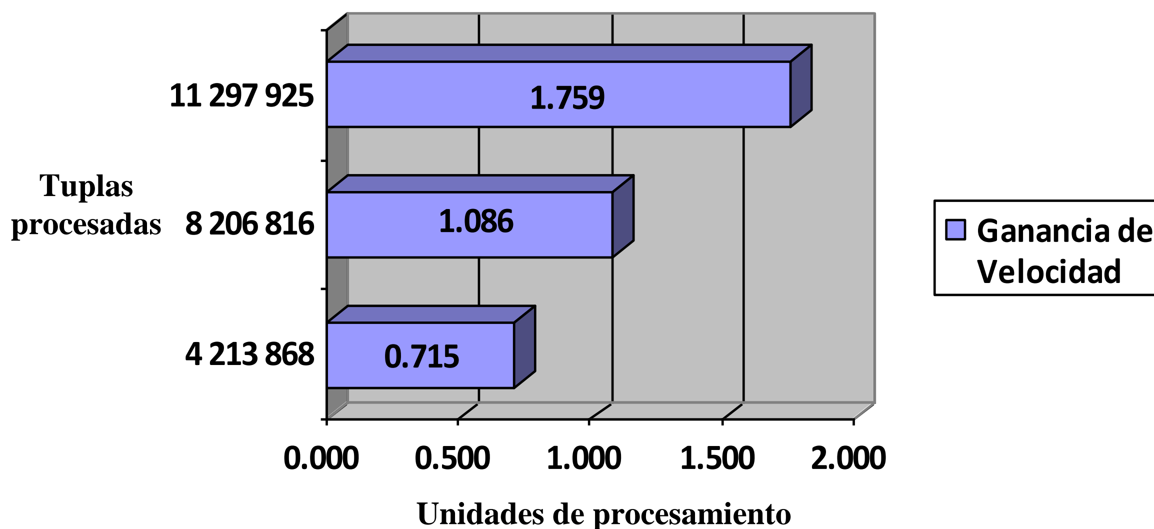

Figura 3.2-Resumen de la ganancia de velocidad del algoritmo paralelo.

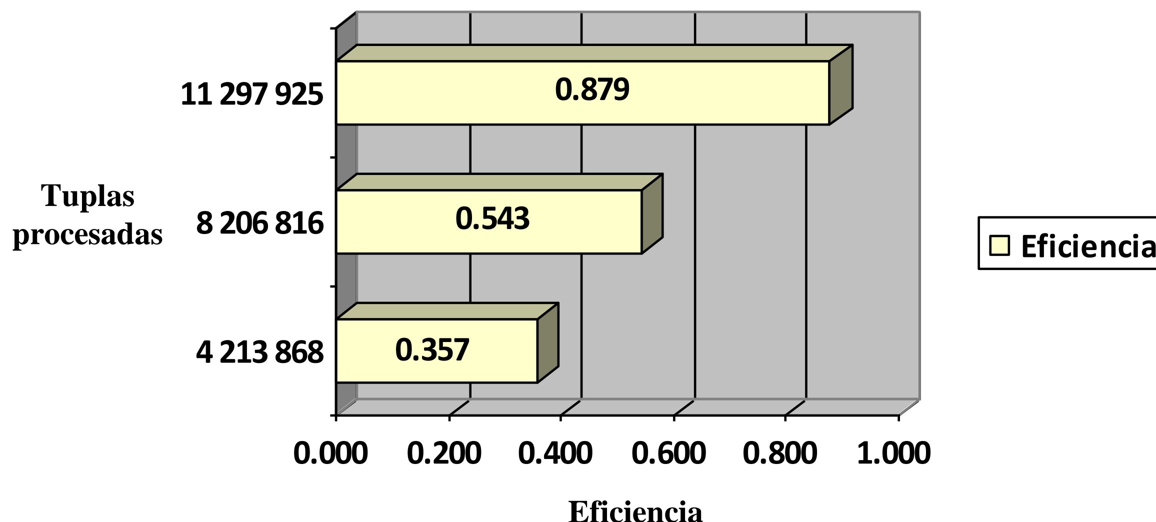

Figura 3.3-Resumen de la eficiencia del algoritmo paralelo.

Dado que el algoritmo implementado solo utiliza dos hilos, y debido a las limitaciones para el trabajo en paralelo con la arquitectura actual de PostgreSQL, se puede descartar la escalabilidad[11] del mismo. Sin embargo en los resultados de las pruebas apreciados en las figuras 3.1, 3.2 y 3.3 se puede apreciar que a medida que ocurre un aumento en la cantidad de tuplas a procesar, se evidencian mejoras en los tiempos de ejecución de las consultas, una ganancia de velocidad significativa con un aumento en el grado de eficiencia en el trabajo paralelo, lo que representa un resultado satisfactorio y una ganancia representativa en términos de rendimiento durante la ejecución de consultas utilizando algoritmos paralelos en PostgreSQL.

3.5 Conclusiones del capítulo 3

En este capítulo se proponen dos algoritmos paralelos para el procesamiento de las tuplas durante la ejecución de una consulta. Una propuesta se basa en el modelo de dos productores-consumidores, variante que no fue implementada en la práctica debido a las limitaciones de la arquitectura de PostgreSQL. La segunda variante es basada en el modelo productor-consumidor clásico, en el cual un hilo produce las tuplas y otro las consume; obteniéndose resultados satisfactorios debido a la reducción del tiempo de ejecución de las consultas. El parámetro eficiencia paralela para el algoritmo

[11] **escalabilidad: tomando una definición informal de escalabilidad como la ganancia de velocidad del algoritmo paralelo al aumentar la cantidad de núcleos computacionales. El algoritmo propuesto, al usar sólo dos hilos de procesamiento, elimina la posibilidad de usar más de dos núcleos por lo que no tiene sentido hablar de escalabilidad**.

tuvo un comportamiento adecuado. Las pruebas realizadas evidenciaron que la ejecución de consultas complejas utilizando PostgreSQL se benefician de la utilización de algoritmos paralelos, aunque con la limitación en este caso que implica la utilización eficiente de solo dos núcleos computacionales, debido al modelo paralelo escogido. Utilizando hilos se logra la distribución de la carga de procesamiento, facilitando y agilizando el procesamiento de los datos computacionales y la velocidad de respuesta del gestor a las peticiones del usuario.

CONCLUSIONES GENERALES

En el presente trabajo se realizó el diseño y la implementación de un algoritmo paralelo que aprovecha la capacidad de cómputo existente en los equipos multinúcleo y/o multiprocesadores durante la ejecución de consultas utilizando el gestor de bases de datos PostgreSQL. Para lograrlo, se realizó un estudio detallado sobre los conceptos fundamentales relacionados con las arquitecturas multinúcleos, la programación concurrente y paralela, algunos de sus modelos de algoritmos, y los procesos e hilos con sus variantes de implementación en los sistemas operativos con *kernel* de tipo Unix. El análisis realizado en el transcurso de la investigación arrojó los siguientes resultados:

- Se utilizó hilos gestionados por la biblioteca *Pthread* para paralelizar parte del proceso de ejecución de consultas en el gestor de bases de datos PostgreSQL y así aprovechar la capacidad de cómputo disponible en equipos de cómputo multinúcleos y/o multiprocesadores.
- Se diseñaron dos algoritmos paralelos basados en el modelo productor-consumidor para paralelizar el procesamiento de tuplas durante la ejecución de las consultas. Una variante con dos hilos procesadores de tuplas, limitada por la arquitectura del gestor para el trabajo en paralelo. La otra variante implementa el modelo clásico donde un hilo productor gestiona las tuplas y otro consumidor las almacena.
- Se implementó un algoritmo paralelo para lograr obtener un mayor rendimiento en la ejecución de los procesos de creación, obtención y almacenamiento de las tuplas durante la ejecución de una consulta compleja en el gestor de bases de datos PostgreSQL.
- En el proceso de validación del algoritmo se realizaron pruebas de tiempo de ejecución, ganancia de velocidad, de eficiencia y de rendimiento. Se probó con consultas altamente costosas que demostraron que la solución cumple satisfactoriamente con el objetivo propuesto. Se redujo el tiempo de ejecución de las consultas con una ganancia velocidad de 1.759 (de un máximo teórico posible de 2) y una eficiencia de procesamiento paralelo aproximado a un ochenta y cinco por ciento para cantidades de tuplas superiores a los once millones.

RECOMENDACIONES

- Realizar un estudio del manejo de memoria de PostgreSQL, para definir el comportamiento de las operaciones con el contexto de memoria cuando se utiliza dentro de varios hilos.
- Estudiar la a arquitectura de PostgreSQL, y crear condiciones que permitan el acceso concurrente de los hilos a las diferentes estructuras de datos.
- Continuar profundizando esta investigación, con el objetivo de buscar variantes paralelas de ejecución del módulo: Planeador/optimizador.

REFERENCIAS BIBLIOGRÁFICAS

[1] Hennessy, J.L. y Patterson, D.A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1996.

[2] Kumar, V., Gramar, A., Grupta A, Kerypis, G. *Introduction to Parallel Computing*. Second Edition, Addison Wesley, 2003.

[3] Tanenbaum, Andrew S. *Modern Operating Systems.* Second Edition*, 2001.*

[4] Marcel Overview. Marcel Overview. [En Línea][Citado el: 3 de diciembre de 2011]. Disponible en: http://runtime.bordeaux.inria.fr/marcel/

[5] Carrera Carrera, Sara. *Adquisición semi-automática del conocimiento: una arquitectura preliminar*. [En línea] [Citado el: 5 de diciembre de 2011.] Disponible en: http://www.grupocole.org/cole/library/ps/Car2007a.pdf.

[6] PgAdmin. PostgreSQL Tools. [En línea][Citado el: 5 de Diciembre de 2011.] Disponible en: http://www.pgadmin.org/

[7] PostgreSQL Cuba | www.postgresql.uci.cu . [En Línea][Citado el: 4 de diciembre de 2011]. Disponible en: http://postgresql.uci.cu

[8] Capel Tuñón, Manuel I. *Multiprogramación con pthreads (hebras POSIX 1003.1c)*. Universidad de Granada, Lenguajes y Sistemas Informáticos, 2003.

[9] Campbell, Colin y Miller, Ade. *Microsoft Press Parallel Programming with Microsoft Visual Cplusplus*. Microsoft Corporation, 2011.

BIBLIOGRAFÍA

1. Tanenbaum, Andrew S. *Modern Operating Systems.* Second Edition*, 2001.*

2. Campbell, Colin y Miller, Ade. *Microsoft Press Parallel Programming with Microsoft Visual Cplusplus*. Microsoft Corporation*, 2011.*

3. Chapman, B., Jost, Gabriele y Van Der Pas, Ruud. *Using OpenMP - Portable Shared Memory Parallel Programming*. Massachusetts Institute of Technology. 2008.

4. Gove, Darryl*. Adison Wesley Multicore Application Programming for Windows, Linux, and Oracle Solaris.* Pearson Education, Inc. 2011.

5. Kumar, V., Gramar, A., Grupta A, Kerypis, G. *Introduction to Parallel Computing*. Second Edition, Addison Wesley, 2003.

6. Tuñón, Capel Manuel, I. *Multiprogramación con pthreads (hebras POSIX 1003.1c).* Universidad de Granada, Lenguajes y Sistemas Informáticos, 2003.

7. The PostgreSQL Global Development Group*. PostgreSQL 9.1.0 Documentation.* The PostgreSQL Global Development Group, 1996-2011*.*

8. The PostgreSQL Global Development Group. PostgreSQL 9.1.4 Documentation, 2011. [En línea]Disponible en: http://www.postgresql.org/docs/9.1/static/index.html

9. Marcel Overview | http://runtime.bordeaux.inria.fr/marcel/ [En Línea][Citado el: 3 de diciembre de 2011]. Disponible en: http://runtime.bordeaux.inria.fr/marcel/

10. PostgreSQL Cuba | www.postgresql.uci.cu . [En Línea][Citado el: 4 de diciembre de 2011]. Disponible en: http://postgresql.uci.cu

11. Blaise Barney. POSIX Threads Programming, 2012. [En Línea]Disponible en: https://computing.llnl.gov/tutorials/pthreads/

12. Vidal, A. *Presente y futuro de la Computación Paralela*, 2000.

13. Oualline, S. *Practical C++ Programming*. O'Reilly, 2002.

14. Bäck, T., Fogel, D., Michalewicz, Z. *Handbook of Evolutionary Computation*. New York and Bristol (UK): IOP Publishing and Oxford University Press, 1997.

15. Kornaros, Giorgios. *Embedded Multi-Core Systems*. New York, London. Fayez Gebali and Haytham El Miligi, University of Victoria, Victoria British Columbia, 2010.

16. El-Rewini, Hesham y Abd-El-Barr, Mostafa. *Avanced Computer and Paralell Processing*. New Jersey. John Wiley & Sons Inc., 2005.

17. Alarcón Medina, José M. *Administración SGBD PostgreSQL*. Cursos PostgreSQL, 2006.

18. Pressman, Roger S. *Ingeniería del Software: Un enfoque práctico*. Sexta Edicion. México, McGraw Hill Higher Education, 2005.

19. Sommerville, Ian, Ingeniería del Software, 7ma edición, editorial Pearson – Addison Wesley, 2005.

20. Ciudad Ricardo, Angel Febe. *Introducción al análisis y modelado de software*. Universidad de Ciencias Informáticas. Mayo, 2011.

21. Ciudad Ricardo, Angel Febe. *El modelado del comportamiento de un sistema*. Universidad de Ciencias Informáticas. Mayo, 2011.

22. Talledo Jimenez, Mónica. *La Guía de los Fundamentos para la dirección de Proyectos (Guía del PMBOK)*. 4ta Edición. Pensylvania, Project Management Institute, Inc., 2008.

23. Kernighan, Brian W., Ritchie, Dennis M. *The C Programming Language*. 2nd Edition. Englewood Cliffs, Prentice Hall, 1988.

24. Oualline, Steve. *Practical C Programming*. 3rd Edition. Sebastopol, Calif., O'Reilly & Associates, Inc, 1997.

25. Hernán Ruiz, Marcelo. *PROGRAMACIÓN C*. Buenos Aires, Argentina, M P Ediciones, 2003.

26. Broquedis, François; Furmento, Nathalie; Goglin, Brice; Wacrenier, Pierre-André; Namyst, Raymond. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, October 2010, 38 (5 - 6): 418-439.

27. Mitchell, Samuel. Advanced Linux Programing. Londres, New Riders, 2001.

28. Edward C Herrmann. Threaded Dynamic Memory Management in Many-Core. University of Cincinnati. Ohio, 2010.

29. The PM2 Team1.*Getting started with PM2*. 2009.