UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ESTUDIO DE DESEMPEÑO DE ALGORITMOS EN ENTORNOS MULTICORE

Tesis

Que para obtener el título de
Ingeniero en Computación

P R E S E N T A:
Pablo José Estrada Murguía

Directora de tesis:	**Ing. Laura Sandoval Montaño**
Miembros del Comité:	**M.I. Jorge Valeriano Assem**
	M.I. César Govantes Saldivar
Suplentes:	**M.I. Rubén Anaya García**
	M.I. Elba Karen Sáenz García

MÉXICO, D.F 2011

Agradecimientos

Agradezco primero que nada a mi madre. Todo esto ha sido posible gracias a ella.

Es importante agradecer también a la Facultad de Ingeniería y a la Universidad por todas las posibilidades, las oportunidades y el apoyo que me han brindado.

A la maestra Laura Sandoval, quien me apoyó completamente durante todo el último año de mi carrera, para completar este trabajo, mi servicio social, y demás actividades. Su apoyo ha sido absolutamente vital e indispensable para que esto ocurriera.

Al profesor Puneet Gupta, de la Universidad de California en Los Angeles, por despertar y guiar mi interés por la arquitectura de computadoras. Él fue la principal fuente de inspiración para este trabajo.

Quiero agradecer también a los profesores Bruce Rosen, de UCLA y Tim Kraft de la Universidad de Stanford, así como a Elaine Wah y Supriyo Chakraborty de UCLA, porque sin que yo mismo lo supiera, me ayudaron a descubrir cosas que permitieron que este trabajo llegara a feliz término.

Gracias también a todos mis amigos.

Índice

Estudio del desempeño de algoritmos en entornos multicore

Capítulo 1. Introducción

De mis motivaciones

El cómputo paralelo es un tema muy interesante. Mi interés surgió temprano en mi carrera, pero fue cultivado en mi clase de Arquitectura de Computadoras. Ahí hablamos de varios factores importantes del desempeño de una arquitectura, como la jerarquía de memoria, pipelines, múltiples núcleos, sistemas de entrada-salida, etc. Eso me hizo preguntarme sobre la programación y los efectos que puede tener la arquitectura de la computadora sobre la programación.

Temas como algoritmos optimizados para el sistema de memoria, arquitectura y sus nuevos avances, entre otros son temas que también me interesan mucho.

La programación para arquitecturas paralelas es un campo muy amplio. Antes de poder escoger un tema preciso, navegué por muchos aspectos de ésta: compiladores paralelizadores automáticos, diseño de arquitecturas de hardware paralelas, inferencia de paralelismo en tiempo de ejecución, análisis de algoritmos, jerarquías de memoria, etc. Tras un semestre de vagabundear en el tema, decidí que quería estudiar el desempeño de algoritmos y programas desarrollados en paralelo.

Para los ingenieros de software, el punto clave entre utilizar la programación serial y empezar a desarrollar en paralelo se encuentra en el desempeño y la productividad: La existencia de técnicas y herramientas que faciliten extraer desempeño de programas y arquitecturas paralelas será un factor que determine la popularización de la programación paralela.

Es por esto que decidí realizar este trabajo que presento a continuación.

Sobre este trabajo

El objetivo de este trabajo es comprender de manera extensa la utilidad de desarrollar algoritmos que exploten paralelismo y las maneras en que éste puede ser utilizado en provecho del software, así como entender cuándo y cómo es conveniente paralelizar software en entornos de multiprocesadores de memoria compartida.

Para estudiar las consecuencias, tanto positivas como negativas de la implementación de versiones paralelas de los algoritmos, se toma en cuenta el comportamiento de dichos algoritmos en tres entornos distintos:

1. La versión serial del algoritmo.
2. La versión paralelizada del algoritmo ejecutada en un núcleo.
3. La versión paralelizada del algoritmo ejecutada en varios núcleos.

Estas tres condiciones permitirán estimar varias características importantes del comportamiento de los algoritmos paralelos.

La tasa del tiempo de ejecución de la versión paralelizada ejecutada en un núcleo contra el de la versión serial es la eficiencia del programa paralelo contra el programa serial; permite estimar el costo extra que se incurre al crear hilos y la planificación en tiempo de ejecución. Con esto es posible observar que los hilos suficientemente largos amortizan el costo de planificación y hasta qué punto éste es un costo en hilos demasiado cortos.

La tasa del tiempo de ejecución de la versión paralela en varios núcleos entre el tiempo de ejecución de la misma en un solo núcleo permite estimar el paralelismo real de la aplicación, y estimar hasta qué punto la ecuación de tasa de desempeño sigue siendo lineal.

Capítulo 2. Estado del arte de la computación paralela

El cómputo paralelo tiene muchas caras. El desarrollo reciente de la programación paralela ha sido empujado por el hardware, pero existen muchas líneas de investigación y muchas áreas de la computación que se conjugan para dar forma al cómputo paralelo.

A continuación se presentan los temas fundamentales para entender la programación de algoritmos paralelos en entornos de multiprocesadores de memoria compartida.

Las revoluciones del hardware

La mayor parte de los algoritmos actuales son secuenciales; es decir que especifican una secuencia de pasos donde cada paso es una operación sencilla. Sin embargo, aunque las computadoras secuenciales habían mantenido una tasa consistente de incremento en su desempeño, con el tiempo este incremento fue volviéndose más costoso, empeorando algunos de los factores, como consumo de energía, complejidad de manufactura, etc. Ya hace algunos años, a los grandes fabricantes de microprocesadores se les acabaron los trucos para seguir aumentando el desempeño de sus procesadores de manera tradicional. Los pipelines más profundos, sistemas de ejecución en desorden, ejecución de múltiples instrucciones por ciclo y el aumento de la frecuencia de operación dejaron de ser efectivos; conforme más complejas se volvían las arquitecturas, su eficiencia disminuía y su consumo de potencia aumentaba, y claro, todo esto bajo la constante presión del mercado que año con año exige sistemas más rápidos. Por este motivo, los ingenieros de hardware tuvieron que buscar nuevas estrategias para seguir aumentando el desempeño de sus procesadores conforme el mercado lo exigía.

El resultado fueron los microprocesadores multicore de propósito general: sistemas de microprocesadores con varios núcleos de ejecución en un solo chip; esto le permitió a los fabricantes aumentar el desempeño de sus sistemas de propósito general y mantener un consumo de potencia en un rango aceptable. [1]

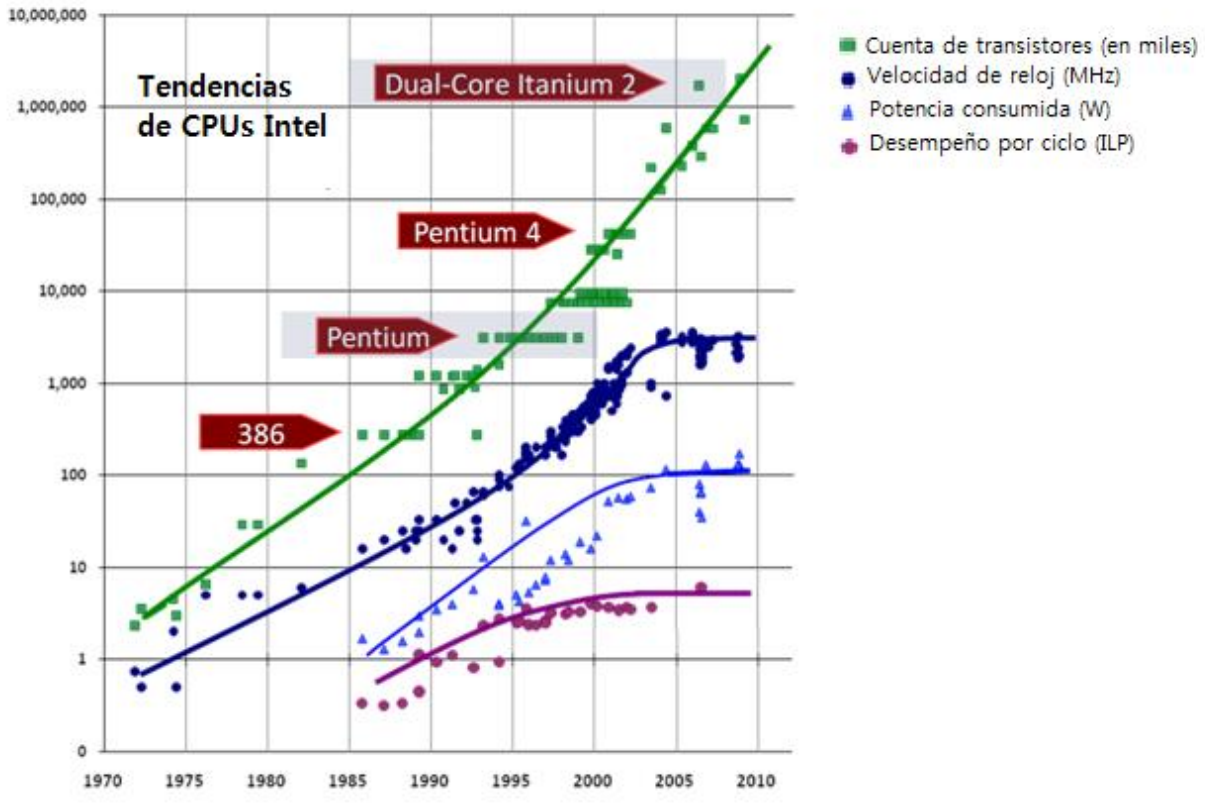

Figura 1. Microprocesadores de Intel hasta agosto de 2009. [2]

En la figura 1 se muestran las gráficas de cuenta de transistores, frecuencia de reloj, potencia consumida y el desempeño por núcleo por ciclo de reloj de los lanzamientos de procesadores de Intel hasta agosto de 2009. Como vemos, entre 2000 y 2005 se alcanzaron barreras de frecuencia, potencia y desempeño que obligaron a los fabricantes a buscar nuevas técnicas para seguir escalando el desempeño de sus microprocesadores.

La desventaja que se trajo con el desarrollo de arquitecturas multicore es que para ejecutar un programa que resuelva un problema de manera eficiente, es necesario utilizar algoritmos que especifiquen operaciones en paralelo. Esto representa un reto para la ingeniería de software: se ha vuelto necesario que los ingenieros de software utilicen explícitamente las ventajas que el hardware les ofrece; sin embargo, el desarrollo de software en entornos paralelos es mucho más complejo que en entornos seriales, y tiene muchas consideraciones que nunca antes fueron necesarias en la programación.

Es interesante observar de la figura 1 que a diferencia de la frecuencia, la cuenta de transistores ha seguido aumentando exponencialmente conforme lo predice la Ley de Moore; esto ha permitido que se agregue cada vez una mayor sofisticación y complejidad así como más núcleos a los nuevos microprocesadores. De hecho, Intel Corp. mantiene un alto ritmo de desarrollo en sus microprocesadores de propósito general a través de su modelo Tick-Tock[3]; este modelo consiste en un ciclo de dos años, donde se introduce una nueva microarquitectura un año, y un nuevo proceso de

transistor que permite refinar la microarquitectura al año siguiente. Este modelo ha resultado en avances muy frecuentes en el hardware y en el software, y un fuerte empuje para el desarrollo de la programación paralela.

Vale la pena comentar que existe otra rama de la computación donde se ha desarrollado el cómputo paralelo de memoria compartida: la computación gráfica.

Resulta que desde hace varios años fue necesario que los fabricantes de tarjetas gráficas incorporaran un gran poder de procesamiento paralelo en sus dispositivos, ya que el *render*, es decir, el dibujo de imágenes y modelos es un proceso muy complejo que requiere gran cantidad de cálculos, que además tienen altos grados de paralelismo. Las unidades de procesamiento gráfico (GPUs) llegan a tener cientos de núcleos mucho más sencillos que los núcleos disponibles en los chips multicore de propósito general, y es común que el chip pueda ejecutar miles de hilos al mismo tiempo, mediante tecnología de *multithreading*, es decir que cada núcleo ejecuta dos, cuatro o más hilos para aprovechar todo el paralelismo posible. [4]

Corrientes en el diseño de procesadores paralelos de memoria compartida

Aunque este trabajo se enfoca a chips multicore en general, es interesante comentar las corrientes existentes entre los arquitectos de hardware para el diseño de multiprocesadores. Existen varias características de los procesadores multicore que cada fabricante adapta a las necesidades de su manufactura y su mercado objetivo. Estas características pueden ser la tasa de potencia/desempeño, características particulares del pipeline, la red de intercomunicación y el sistema de memoria.

- La tasa potencia/desempeño

Existen muchos dispositivos limitados en su potencia y desempeño. Por ejemplo, teléfonos celulares que reproduzcan video tienen un límite estricto de potencia, pero también deben alcanzar ciertas características de desempeño. Estas limitantes han provocado que los fabricantes no sólo busquen alto desempeño, sino también pagar un consumo de potencia aceptable por ese alto desempeño. Esto ha invitado a que muchos fabricantes tengan líneas de bajo consumo de energía, como los procesadores Atom de Intel y otras líneas de procesadores para sistemas embebidos.

- Características del pipeline

Sin duda, las características del pipeline de ejecución son uno de los campos más variantes de las arquitecturas multicore. Según las necesidades del fabricante y de su mercado, éste puede decidir preparar el pipeline para mejorar su desempeño en ciertas áreas. Aquí es donde la comparación del cómputo gráfico contra el cómputo de propósito general da resultados interesantes.

Intel es el mayor ejemplo de manufactura de procesadores de propósito general de alto desempeño. En este campo, se busca explotar el paralelismo a nivel de hilo con múltiples núcleos; pero esto sin perder desempeño en la ejecución de cada hilo: Intel busca obtener el máximo desempeño de ejecución por hilo, así que incluyen capacidades de cómputo *single instruction, multiple data*, que permite ejecutar la

misma operación sobre cada elemento en largos vectores de información. También utilizan motores de ejecución en desorden y múltiples unidades aritmético-lógicas para extraer todo el paralelismo a nivel de instrucción que les sea posible. Los pipelines de los procesadores de escritorio de Intel son líneas de ensamblaje muy complejas y de muy alto desempeño y consumo de energía.

Por el otro lado se tienen los procesadores de cómputo gráfico fabricados por NVIDIA. Estos procesadores buscan explotar paralelismo de una cantidad masiva de hilos más sencillos y más optimizados. Esto ha permitido que se enfoquen en procesadores con cantidades masivas de núcleos sencillos, de ejecución en orden, pero eso sí, con multihilado para ocultar el bajo desempeño de aquellos hilos que tengan demasiada comunicación con la memoria.

- Características de la red de comunicación y la jerarquía de memoria

La red de comunicación y la jerarquía de memoria son partes distintas de un microprocesador, pero especialmente con los multiprocesadores de memoria compartida, tienen una relación muy estrecha.

Dado que se tienen varias unidades de procesamiento que funcionan simultáneamente, y cada una cuenta con sus propios niveles de cache individuales, un problema importante de estas arquitecturas es mantener la consistencia de memoria. Se busca mantener una sola imagen de memoria entre todos los procesadores, que están accediendo a ella frecuentemente, y tomando segmentos de ésta para moverlos a su cache; esto permite que existan inconsistencias cuando un procesador modifica el segmento de memoria en su propio cache, pero no actualiza el resultado en la memoria principal, y los fabricantes deben tomar una decisión para tratar estos problemas. Dado que los procesadores se comunican con la memoria y entre sí a través de la red de comunicación, la consistencia de memoria y la red están fuertemente asociadas.

La forma más sencilla de red de intercomunicación es el bus. Esto permite que la comunicación sea muy sencilla, ya que todos los procesadores están enterados de lo que ocurra en memoria y qué procesador es el que posee cierta localidad de memoria. El problema con el bus es que su ancho de banda se ve limitado conforme más procesadores hay conectados a éste. De hecho, cuando se habla de más de 8 núcleos, no es común encontrar buses de intercomunicación.

Las redes de intercomunicación pueden ser muy complejas; el algoritmo de consistencia se vuelve más complejo conforme la red aumenta, algunos fabricantes deciden no implementar ningún algoritmo de consistencia. Esto es común en GPUs, que aprovechan el paralelismo de datos para salvar el modelo de memoria sin consistencia.

Existen otras direcciones en que la tecnología ha seguido avanzando. La cadena de valor de los microprocesadores se realimenta con las necesidades de sus mercados y crea nuevas tendencias. Las anteriores son las más sobresalientes en los últimos años.

Modelando una computadora paralela

Los algoritmos secuenciales típicamente son modelados con pseudocódigo, y en una máquina de acceso aleatorio simple, como en la figura 2. En este modelo, cada operación requiere de un solo paso: operaciones aritméticas, lógicas, saltos o de acceso a memoria toman un paso.

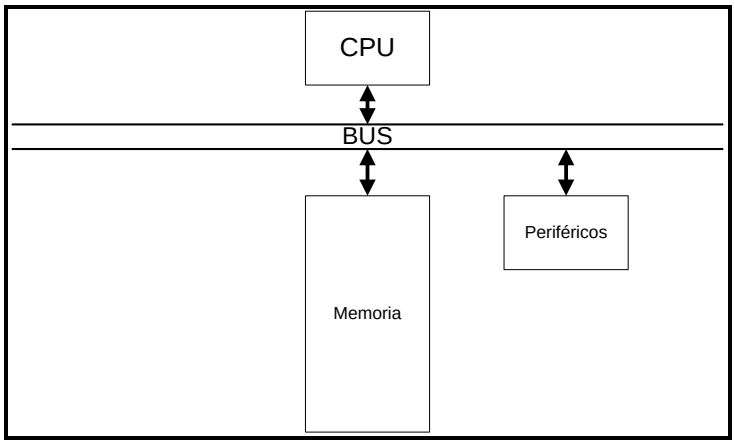

Figura 2. Modelo de una computadora serial. [5]

Para modelar computadoras paralelas, usualmente se requieren modelos más complejos, debido a que en general, la organización de las computadoras paralelas es mucho más compleja que la de las computadoras seriales comunes. Hay, de hecho, mucha investigación dedicada a encontrar modelos precisos y útiles para el cómputo paralelo. No hay ningún estándar común que se utilice en la academia o en la industria. [6]

El modelo que se utiliza en este trabajo es el modelo PRAM (*Parallel Random Access Machine*), que no es más que una ampliación al modelo de máquina de acceso aleatorio. El modelo PRAM consiste de varios procesadores que se comunican mediante un bus o una red de intercomunicación y comparten la misma memoria, como se ve en la figura 3.

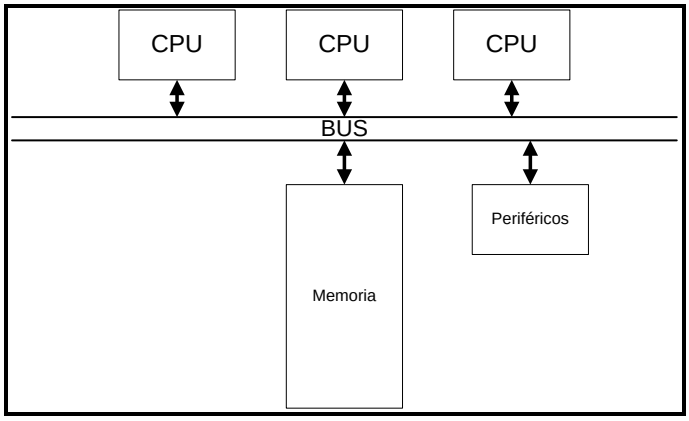

Figura 3. Modelo PRAM de una computadora paralela. [6]

En un modelo PRAM se asume que todas las operaciones toman un tiempo unitario para ejecutarse, tanto las operaciones aritméticas, lógicas, saltos y accesos a memoria, además se entiende que N operaciones son realizadas en un solo paso, donde N es la cantidad de procesadores; por otro lado, también se considera que cuando un procesador escribe en cierta posición de memoria, ésta es reservada para dicho procesador, mientras que al leer, se permite que varios procesadores accedan la misma posición de memoria. Este modelo es conocido como PRAM de escritura exclusiva y lectura concurrente. También es posible utilizar modelos de lectura exclusiva; sin embargo, para esta tesis, se considera el modelo de lectura concurrente y escritura exclusiva.

Este modelo de lectura concurrente y escritura exclusiva permite modelar una condición particular de los programas paralelos que crea una dificultad para el programador: las carreras de datos. Una carrera de datos ocurre cuando dos hilos acceden a la misma posición de memoria, y al menos uno de ellos la modifica. Dado que la planificación de los hilos de ejecución es impredecible, no es posible saber si un hilo va a acceder antes o después de otro al mismo segmento de memoria, y esto significa que en condiciones iguales, el programa puede tener resultados distintos; es decir que las condiciones de carrera son indeseables en la programación, ya que producen resultados no determinísticos en el programa y complican muchísimo depurar un algoritmo paralelo.

A continuación se presenta un ejemplo sencillo donde puede ocurrir una carrera de datos:

```
int A=0;
principal(){
      en paralelo:
            suma();
            divisible();
}
divisible(){ while(1) if ( A%2 == 0 ) print(A); }
suma(){ while(1) A = A + 1; }
```

En el ejemplo anterior, dos funciones acceden a la misma variable en paralelo. Esto significa que la salida del programa no es determinística, porque no es posible predecir en qué orden y por cuánto tiempo un hilo sería dueño del dato actual. A continuación se muestran dos ejemplos diferentes de salidas posibles del mismo programa:

0	2
0	12
0	18
4	20
8	20
10	22

Como se puede ver, las carreras de datos producen resultados impredecibles en el programa y por eso es necesario evitarlas.

Un problema de los modelos PRAM por el que son bastante criticados es que ninguna máquina real tiene acceso a memoria con tiempo unitario; sin embargo, no debemos olvidar que el propósito de un modelo abstracto es asistir en el diseño de un algoritmo, no el modelar exactamente una máquina real; es decir que si un algoritmo basado en el modelo PRAM puede ser implementado en un programa real y eficiente, el modelo sirve su propósito.

En el modelo de la figura 3, los procesadores y la memoria se interconectan mediante un bus; sin embargo, existen diversas topologías de red de intercomunicación que son mucho más complejas[1].

¿Pero… qué es el desempeño?

Existen varios métodos en que el desempeño de una computadora puede ser medido; desde la frecuencia del reloj, la cantidad de instrucciones ejecutadas por unidad de tiempo, etc. Sin embargo, la manera más práctica y útil de definir el desempeño es mediante el tiempo de ejecución: aquella computadora o programa que pueda ejecutar una tarea más rápido, es la que tendrá el mayor desempeño. [5]

Notación y terminología

En este trabajo hay terminología y algunas literales que se utilizan; a continuación se aclara explícitamente el significado de algunos términos utilizados a lo largo del texto:

- **T**. La literal T se refiere al tiempo de ejecución de un programa; el subíndice especifica las condiciones en que este tiempo es medido.

- **Tserial**. Esto es el tiempo de ejecución de la versión serial de un programa. La versión serial de un programa se ejecuta en un solo núcleo, y es tomada como referencia para otras cantidades importantes.

- **Tp**. Es el tiempo de ejecución de la versión paralela de un programa, donde **p** es la cantidad de hilos de ejecución utilizados para ejecutarlo.

- **T1**. Es el tiempo de ejecución de la versión paralela de un programa utilizando un solo hilo de ejecución; esta cantidad es importante para medir la efectividad del algoritmo.

- **Efectividad**. Un algoritmo paralelo es efectivo si el trabajo que realiza no es mucho mayor que la versión serial de dicho algoritmo. Para medir la efectividad cuantitativamente se utiliza el cociente **T1/Tserial**; esto nos da una noción del trabajo extra realizado por el algoritmo paralelo, respecto de su versión serial.

[1] Revisar [5, 6. Network Topologies] para más información sobre las redes de intercomunicación de distintos multiprocesadores.

- **Aceleración**. En este trabajo se busca medir qué tanto se acelera la ejecución de un programa al utilizar más hilos para ejecutarlo. La aceleración es una función, dada por el cociente **T1/Tp**, donde **p** es la cantidad de hilos de ejecución utilizados. Ésta es una función muy importante en este trabajo.

Por convención, en este trabajo, cuando se diga que el desempeño de un programa X es N veces mayor que el de otro programa Y, significa que el tiempo de ejecución de Y es igual a N veces el tiempo de ejecución del programa X, es decir:

$$\frac{T_Y}{T_X} = N$$

Por otro lado, cuando se hable de cantidades porcentuales, entonces la diferencia en desempeño es aditiva. Por ejemplo, al decir que un programa X tiene un desempeño N por ciento mayor que otro programa Y, esto significa que:

$$\frac{T_Y}{T_X} = \frac{100 + N}{100}$$

Es importante tener en cuenta estas convenciones a la hora de leer los resultados de los experimentos del trabajo, ya que este trabajo está enfocado en comparar los tiempos de ejecución de algoritmos con distintos niveles de paralelismo y así entender las mejoras de desempeño que provee el paralelismo.

Técnicas de desarrollo de algoritmos paralelos

Existen diversas técnicas que permiten desarrollar algoritmos paralelos según el problema que se esté atacando. Algunas son similares a ciertas técnicas utilizadas para desarrollar algoritmos secuenciales y otras son especiales para algoritmos paralelos. Estas técnicas permiten escribir implementaciones eficientes de los algoritmos, y aprovechar al máximo el paralelismo.

- Rediseño de estructuras de datos

Una estrategia útil y poco obvia de algoritmos paralelos es utilizar estructuras de datos que permitan explotar el paralelismo mejor. La mayor parte de las estructuras de datos que se utilizan en la actualidad no permiten aprovechar el paralelismo eficientemente, o provocan condiciones de carrera de datos, y problemas en la coherencia de memoria. [6]

A diferencia de las arquitecturas de un solo núcleo, la coherencia de memoria es un problema especialmente problemático en arquitecturas paralelas de memoria compartida. Dado que existen varias unidades que pueden acceder a las mismas localidades de memoria; y dado que la jerarquía de memoria permite que a lo largo de los niveles de memoria existan varias copias de la información guardada en una localidad, es necesario crear un sistema que asegure que todos los procesadores vean la misma imagen de memoria; es decir que cuando un procesador modifique la información almacenada en una localidad de memoria, es necesario que todos los procesadores tengan un mecanismo para conocer el valor más reciente guardado en dicha localidad. Las arquitecturas manejan la coherencia de

memoria de distintas maneras según las prioridades que se tengan en cuenta para su diseño. En el manejo de la coherencia de memoria también es importante el modelo de consistencia, que tiene que ver con el orden en que se realizan las operaciones en la memoria.[2]

- Paralelización de estructuras

Un ejemplo muy común de una estructura de datos poco eficiente para su uso paralelo es cuando se almacena una secuencia en una lista ligada; esta estructura obliga al programa a recorrer elemento por elemento para atravesarla, y si es necesario operar en ellos, o buscar en la lista, esto genera una sección secuencial innecesaria; debido a que el recorrido de una lista es un algoritmo serial. Si en vez de utilizar una lista ligada, se utilizan arreglos con direcciones calculables, cada hilo de ejecución puede tener acceso directo a cada uno de los elementos de la secuencia de manera inmediata; sólo es necesario hacer el cálculo de la dirección de memoria basado en el índice.

Los árboles también son estructuras que en general permiten implementar algoritmos paralelos con pocas dificultades, al existir varios caminos independientes en un árbol, es fácil recorrer cada uno en paralelo sin interferencia entre las diversas tareas.

- Creación de reductores

Un problema común de la programación paralela es concentrar los resultados de varios cálculos realizados en paralelo en una sola variable. Este proceso es conocido como reducción, y es muy relevante para los desarrolladores.

En muchos casos, los reductores se construyen creando una variable por cada hilo, y posteriormente sumando todos esos valores explícitamente tras la ejecución del programa. Este método es útil, pero crea gasto en la etapa de concentración de la suma. Existen APIs que proveen de estructuras de datos que funcionan en conjunto con el sistema de ejecución para que las reducciones sean realizadas automáticamente por el sistema, y no manualmente por el programador. Esto crea cierto gasto, que aún es mucho menor al que se crearía con reducciones manuales.

- Rediseño de los algoritmos

La reingeniería de estructuras de datos es complementada por la reingeniería de algoritmos. Este proceso permite crear software escalable y funcional.

- Divide y conquista

Esta técnica es también utilizada para el diseño de algoritmos secuenciales; consiste en dividir un problema grande en otros problemas más pequeños, más sencillos de resolver. Esta técnica juega un rol eminente en algoritmos paralelos, ¡ya que la solución de cada subproblema puede ser obtenida en paralelo! De hecho, es común que aquellos algoritmos diseñados para máquinas secuenciales con la técnica de divide y conquista tengan altos grados de paralelismo.

[2] Para más información sobre la coherencia de memoria y los modelos de consistencia, ver [1] y [4].

Como nota, cabe mencionar que para que el algoritmo sea altamente escalable, también es conveniente paralelizar la división de los problemas y la unión de las soluciones; si no, el componente serial del algoritmo se vuelve una limitante del paralelismo que puede ser aprovechado.

Un ejemplo excelente de un algoritmo de este tipo es el ordenamiento por mezcla: la lista a ordenar es dividida entre dos; cada una de las dos listas es ordenada y luego las dos listas son mezcladas en orden. Este algoritmo se resuelve recursivamente, y su implementación es la siguiente:

```
ordenar_por_mezcla(arreglo)
      si ( arreglo.tamaño() == 1 )
            devolver arreglo;
      de lo contrario
            en paralelo (
                  arreglo a =
            ordenar_por_mezcla(arreglo.mitadInferior()),
                  arreglo b =
            ordenar_por_mezcla(arreglo.mitadInferior())
                  );
            devolver mezcla_en_orden(a, b);
```

Complejidad de algoritmos y notación Big-O

El análisis de algoritmos es una rama de la ciencia de la computación que se dedica a estudiar y tratar de medir el comportamiento de los algoritmos. El análisis de un algoritmo procura determinar la cantidad de recursos que el algoritmo utilizará al ser ejecutado, tales como tiempo y memoria, principalmente.

El análisis de la complejidad permite estimar teóricamente el comportamiento de los algoritmos. Dado que en este trabajo se estudia el comportamiento en tiempo de algoritmos paralelizados, es importante conocer el método de análisis teórico de la complejidad temporal de un algoritmo.

En el análisis de algoritmos, el análisis de tiempo de ejecución de un algoritmo procura anticipar el tiempo de ejecución de un algoritmo en función de las características de su entrada. Normalmente, se estima en función del tamaño de la entrada; por ejemplo, la complejidad temporal de un algoritmo de ordenamiento se calcula en función del tamaño del arreglo que se quiere ordenar, mientras que la complejidad temporal de un algoritmo que calcula el factorial de un número se calcula en función del número para el que se quiere calcular el factorial.

El análisis de tiempo de ejecución de un algoritmo normalmente es realizado estudiando la estructura del algoritmo y de su código. En ocasiones es necesario realizar asunciones para simplificar el análisis y obtener una función que estime razonablemente la complejidad temporal del algoritmo.

Finalmente, para estudiar la complejidad temporal de un algoritmo, se utiliza notación asintótica. La notación Big-O permite expresar el concepto de orden de crecimiento de la complejidad de un algoritmo según la entrada. Básicamente se toman los términos de orden superior de la complejidad del algoritmo, y se desprecian los de orden inferior y las constantes.

A continuación se presenta el código de una función que calcula la suma de los primeros n números naturales:

```
int calcularSuma(int n){
      int suma = 0;
      for (int i = 1; i <= n; i++){
            suma += i;
      }
      return suma;
}
```

Queremos calcular la cantidad de operaciones que este código realiza; considerando que la suma, la comparación y la asignación cuentan cada una como una sola operación. La creación de las variables `suma`, e `i` son dos operaciones. Luego, por cada vez que se entra al ciclo `for`, se realizan 3 operaciones: Comparar `i` contra `n`, hacer `suma += i` e incrementar `i`. Finalmente se devuelve el valor de `suma`. Por lo tanto, este programa realiza `3+3n` operaciones. Por lo tanto, se dice que para este algoritmo, `T(n) = 3 + 3n`.

Como es posible ver en la ecuación de la complejidad temporal de dicho algoritmo, el término de mayor orden es `3n`, así que se dice que `T(n) = O(n)`, ya que conforme `n` crezca y se haga muy grande, la cantidad de operaciones que realizará el algoritmo aumentará linealmente con el valor de la entrada.

Ahora se presenta un ejemplo un poco más complejo, que sirve para calcular la complejidad de algoritmos recursivos. Se utiliza el algoritmo de ordenamiento por mezcla que se presentó anteriormente, pero en serie:

```
ordenar_por_mezcla(arreglo)
      si ( arreglo.tamaño() == 1 )
            devolver arreglo;
      de lo contrario
            arreglo a = ordenar_por_mezcla(arreglo.mitadInferior())
            arreglo b = ordenar_por_mezcla(arreglo.mitadInferior())
      devolver mezcla_en_orden(a, b);
```

En este algoritmo, hay una cantidad constante de operaciones en las comprobaciones condicional, que puede ser expresada como `O(1)`, esto significa que la cantidad de operaciones es constante. Luego hay dos llamadas recursivas, en las que se ordena la mitad inferior del arreglo y luego la mitad superior; dado que las llamadas son recursivas, el tiempo que estas llamadas toman puede ser expresado como `T(n/2)`. Finalmente, se utiliza la función `mezcla_en_orden(a, b);` sobre las dos mitades. Una función `mezcla_en_orden(a, b);` puede tener una complejidad temporal lineal, es decir, `O(n)`.

La suma de operaciones por la complejidad de cada uno de los segmentos de código de esta función es la siguiente:

$$T(n) = 2T(n/2) + O(n) + O(1)$$

Por las propiedades de la notación asintótica[3], un término de orden lineal absorbe a un término de orden constante, de manera que la ecuación puede ser presentada de la siguiente manera:

$$T(n) = 2T(n/2) + O(n)$$

Para resolver esta ecuación y obtener una sin recursión, es posible utilizar el método maestro para la obtención de la complejidad de un algoritmo. Este método permite obtener el resultado de una ecuación con una recursión como la anterior, y puede ser revisado en [3]. Al aplicar el método maestro, se obtiene que la complejidad temporal del algoritmo de ordenamiento por mezcla es:

$$T(n) = O(n \log n)$$

La complejidad temporal de algoritmos es un tema nuclear de este trabajo, y vale la pena revisar los conceptos antes de seguir avanzando. Si lo presentado hasta ahora no es suficiente, se recomienda remitirse a un libro de algoritmos.

El análisis de los algoritmos paralelos: Trabajo y profundidad

Dado que existen muchas formas y organizaciones distintas de computadoras paralelas, para modelar los algoritmos que se ejecutan en éstas se ha vuelto mucho más común utilizar un método que en vez de enfocarse en la máquina, se enfoca en el algoritmo. Estos modelos son conocidos como modelos de *trabajo-profundidad*. Los modelos de *trabajo-profundidad* se enfocan en el número total de operaciones que un programa realiza, y la cadena más larga de dependencias entre ellas; se calculan en función del tamaño de la entrada o de su valor: el *trabajo* W es el número de operaciones que se realizan, también es conocido como el tiempo que tardaría el algoritmo en ejecutarse en un solo procesador (T_1), y la *profundidad* D es la cadena más larga de dependencias entre estas operaciones. Algunos se refieren a la profundidad como la longitud de la ruta crítica, o el tiempo que tardaría el algoritmo en ejecutarse en una cantidad infinita de procesadores (T_∞). Con esto se calcula la razón *P=W/D*, que se le llama *paralelismo* del algoritmo.

Dada la definición de W y D, resulta la conclusión que en una máquina con recursos paralelos infinitos, el tiempo de ejecución del algoritmo es del orden de la profundidad de éste $T_\infty(n) = \theta(D(n))$; mientras que en una máquina con un solo procesador, el tiempo de ejecución es del orden del trabajo, es decir que $T_1(n) = \theta(W(n))$. [7]

Respecto del algoritmo de ordenamiento por mezcla es posible calcular su trabajo y su profundidad revisando el código. Esta implementación del algoritmo tiene una región serial en la mezcla y una región paralela en el ordenamiento de cada una de las dos mitades. Si consideramos que la mezcla es $O(n)$, la

[3] Dichas propiedades pueden ser revisadas en cualquier libro de algoritmos. Por ejemplo:
Cormen, T. H. , Leiserson, E. C., Rivest, R. L., Stein, C. (2009). Introduction to Algorithms. (3a. ed.) EE. UU.: MIT Press.

profundidad de esta implementación es $D(n) = D\left(\frac{n}{2}\right) + n$, es decir $D(n) = O(n)$. En una máquina con recursos paralelos infinitos, entonces $T(n) = O(n)$.

La versión serial de este algoritmo tiene $T(n) = 2T\left(\frac{n}{2}\right) + n$, es decir $T(n) = O(nlogn)$, así que la mejora en desempeño entre el algoritmo serial y el paralelo es $O(logn)$. Por lo tanto, aunque el algoritmo es eficiente y escalable, la mezcla limita su desempeño y su escalabilidad.

Existen varios tipos de modelos para expresar el trabajo y la profundidad de un algoritmo; hay modelos gráficos, como circuitos y grafos; y aquellos basados en lenguaje, como los modelos basados en pseudocódigo.

Una manera de expresar un algoritmo paralelo observando su ejecución se conoce como diagrama DAG (*Directed acyclic graph*); donde cada vértice es una cantidad fija de instrucciones a ejecutar. Los DAG son diagramas que facilitan analizar el flujo de un algoritmo, y son sencillos de realizar[4].

En los modelos basados en pseudocódigo, el trabajo y la profundidad se calculan según el trabajo asociado a cada sentencia de pseudocódigo. Por ejemplo, al llamar dos funciones en paralelo, el trabajo es la suma del trabajo de cada una de las funciones, y la profundidad es el máximo de las profundidades de las funciones.

En este trabajo se utilizan los modelos basados en lenguaje para analizar los algoritmos. Esto permite analizar directamente el código de cada algoritmo y asociar costos constantes a cada una de las operaciones.

Planificación de hilos

Hasta ahora, el modelo de trabajo-profundidad no había puesto de manifiesto mucho sobre el tiempo de ejecución de un algoritmo, ya que sólo permite estimar el tiempo de ejecución en procesadores de un solo núcleo, o de una cantidad infinita de núcleos. Para poder estimar el tiempo de ejecución de un algoritmo, es necesario estudiar el planificador de ejecución de los hilos.

Normalmente el planificador existe en tiempo de ejecución, en el sistema operativo o el API[5]; y su objetivo es asignar los recursos de cómputo disponibles a los distintos hilos de ejecución. El tipo más sencillo de planificador es conocido como *greedy*, o *codicioso*; este planificador asigna cada recurso disponible al primer hilo que lo solicite. Esta estrategia de planificación no es óptima, pero es aceptable; de hecho la planificación óptima en multiprocesadores es un problema NP-completo; esto significa que el tiempo de ejecución para cualquier solución conocida de este problema es no determinístico y polinomial; y no hay ninguna manera conocida, eficiente y rápida de resolver el problema.

[4] Para más información sobre DAGs, se recomienda revisar Christofides, Nicos (1975), *Graph theory: an algorithmic approach*, Academic Press, pp. 170–174.

[5] API son las siglas de *Application Programming Interface*, o Interfaz de Programación de Aplicaciones. Una API es un set de bibliotecas con métodos y recursos construidos previamente para ofrecer una capa de abstracción al desarrollador.

Del estudio de la teoría de colas es posible derivar un corolario que demuestra que el paralelismo de un algoritmo es el límite de núcleos en que dicho algoritmo puede ser ejecutado, con un aumento lineal en el desempeño por cada nuevo núcleo. [7] Es decir:

$$T_C = \frac{T_1}{c}$$

Donde:

- T_C : El tiempo de ejecución en C núcleos iguales.
- T_1 : El tiempo de ejecución en un solo núcleo.
- c : La cantidad de núcleos disponibles para ejecutar el algoritmo, que debe ser menor o igual al paralelismo $P = \frac{W}{D}$ del algoritmo.

Este corolario será muy útil al analizar teóricamente las versiones paralelas de los algoritmos para predecir su comportamiento en ambientes de varios núcleos.

En adelante, me referiré a esta relación como la ecuación de tasa de desempeño lineal.

Uno de los detalles sobre el modelo de trabajo-profundidad es que no toma en cuenta factores físicos reales como los hilos compartiendo recursos (éste es un caso importante del HyperThreading), la intercomunicación de procesadores y el comportamiento de la jerarquía de memoria; estos factores externos al modelo trabajo-profundidad pueden afectar el comportamiento de los programas de maneras que el modelo de trabajo-profundidad no puede predecir.

APIs y lenguajes de programación paralela

Con la invasión de las arquitecturas multicore a todas las ramas del mercado, desde el cómputo de escritorio hasta el procesamiento de señales, surgió una demanda de tecnologías de software que permitieran aprovechar los recursos de una arquitectura multicore y a su vez mantener la productividad del programador. En los años 90, dos estándares de programación paralela dominaron el panorama de las herramientas de software para entornos paralelos: OpenMP y MPI. [8]

MPI, que significa *Message Passing Interface*, es un modelo de programación a través de comunicación entre procesos mediante mensajes, y aún conserva un lugar en la programación paralela; sin embargo, su curva de aprendizaje es demasiado brusca, y no vale la pena el esfuerzo para que un programador pueda empezar a desarrollar aplicaciones efectivas.

En cambio, OpenMP se ha convertido también en un estándar *de facto* de desarrollo de aplicaciones que exploten paralelismo. Su facilidad de uso y su simple curva de aprendizaje lo hacen ideal para el programador. Las versiones 2.5 e inferiores de OpenMP permitían explotar el paralelismo *estructurado* de las aplicaciones de manera sencilla y efectiva, es decir que OpenMP 2.5 tenía un modelo que explotaba paralelismo en estructuras como arreglos y ciclos *for*; sin embargo, conforme se ganó experiencia, fue obvio que también era necesario explotar paralelismo proveniente de estructuras más irregulares y dinámicas, como listas ligadas y ciclos *while*. La versión 3.0 de OpenMP incluyó

herramientas para explotar paralelismo dinámico, basado en generación dinámica de tareas; el modelo basado en tareas permite distribuir trabajo a los hilos de manera dinámica. [9]

En el campo de la computación gráfica también ha habido innovación y desarrollo de APIs y lenguajes de programación paralela. El ejemplo más sobresaliente es CUDA (*Compute Unified Device Architecture*), que incluye una extensión del lenguaje C y un planificador en tiempo de ejecución; fue desarrollado por NVIDIA y fue la primera tecnología importante que propuso utilizar GPUs para cómputo de propósito general. CUDA representa al GPU como un set jerárquico de núcleos con bloques de miles de hilos de ejecución, donde cada bloque de hilos comparte un segmento de memoria. El modelo de CUDA permite explotar paralelismo a nivel de datos y de tarea, y ha sido usado efectivamente en el desarrollo de algoritmos que explotan niveles masivos de paralelismo. [8]

Sobre la plataforma Cilk++

En este trabajo de tesis se utiliza el entorno de Cilk++ para el desarrollo de los algoritmos paralelos. Esto debido al enfoque particular que tiene este estudio en la arquitectura x86 y sus versiones de 64 bits, además de su alta eficiencia y simplicidad.

La plataforma Cilk, desarrollada inicialmente por Cilk Arts Inc. en MIT, y posteriormente adquirida por Intel en 2009 [10], es una extensión de C y C++ para escribir programas paralelos para sistemas multicore. La plataforma incluye un compilador con soporte para GCC de Linux y el compilador de C++ de Microsoft, así como un detector de condiciones de carrera y un sistema integral de tiempo de ejecución. Cilk++ agrega únicamente tres palabras clave al lenguaje C++ estándar.

Cilk++ fue una plataforma pionera en el paralelismo de tareas. Esto quiere decir que cuando el programador expresa paralelismo en un programa en Cilk++, no está ordenando que se cree un hilo de ejecución, sino que se expresa la posibilidad de realizar una tarea en paralelo. Cuando un programa en Cilk++ es ejecutado, éste lanza una cantidad fija de hilos de ejecución nativos, que por default son la cantidad de núcleos lógicos disponibles en el procesador. Al ir creando nuevas tareas, éstas son ejecutadas dinámicamente por los distintos hilos de ejecución disponibles. Esto permite que el trabajo sea más justamente balanceado entre los hilos disponibles, pues una tarea no está asignada a cierto hilo, sino que es ejecutada en cuanto alguno de los hilos la alcance en la cola de tareas por ejecuta; esto se logra porque Cilk++ utiliza un planificador de ejecución de “robo de trabajo”, cuya estrategia consiste en que cada uno de los hilos de ejecución realice el trabajo que tiene asignado, y una vez que uno de los hilos ha terminado el trabajo que se le ha asignado, selecciona uno de los otros aleatoriamente, y de ser el caso, ‘roba’ tareas de ése, si es que no ha terminado de realizar el que se le asignó. Se ha demostrado teórica y prácticamente que la ecuación de tasa de desempeño lineal se mantiene también en el sistema de tiempo de ejecución de Cilk++.

Dado que Cilk++ utiliza un planificador *online,* es decir que funciona en tiempo de ejecución; y además crea tareas dinámicamente, existe cierto costo implícito por la planificación y por la creación de las tareas. Esto significa que un programa serial en C o C++ puro funciona más rápido que un programa concurrente en Cilk++ ejecutándose en un solo procesador. Sin embargo, se ha demostrado que el costo

implícito del planificador es despreciable cuando cada tarea es suficientemente larga, y el tiempo invertido creando la tarea y planificando su ejecución pierde relevancia ante un flujo de instrucciones suficientemente largo.

Para acceder a más información sobre la plataforma Cilk++ y detalles particulares de su modelo de programación, se recomienda revisar [10], [11] y [12].

Sobre el Manycore Testing Lab de Intel

En este trabajo, la plataforma donde son ejecutados y evaluados los programas que se desarrollaron es el Manycore Testing Lab. [13]

Intel Corporation, como uno de los mayores evangelistas de la programación paralela, pone a la disposición de los miembros de su comunidad académica el Manycore Testing Lab (MTL). Este laboratorio es un conjunto de servidores y máquinas en línea mediante las cuales es posible experimentar con programas paralelos en un entorno de 32 núcleos.

El MTL cuenta con cuatro microprocesadores tipo Xeon x7560 [14] con 8 núcleos cada uno, que se comunican a través del bus principal. Estos microprocesadores cuentan con arquitectura Nehalem; las arquitecturas Nehalem cuentan con una interconexión llamada Quickpath, además de tener integrada la tecnología HyperThreading.

Quickpath es la red de interconexión de Intel que fue lanzada con la microarquitectura Nehalem; Quickpath provee a cada núcleo del procesador con un canal exclusivo y directo a todos los demás núcleos dentro del procesador. Esto permite que la intercomunicación realizada dentro de cada chip sea muy rápida. [15]

Por otro lado, la tecnología HyperThreading es la implementación de Intel del multihilado simultáneo. Esto significa que un solo núcleo del procesador puede ejecutar hasta dos procesos al mismo tiempo compartiendo recursos. Es decir que el MTL cuenta con 32 núcleos que pueden ejecutar hasta 64 hilos al mismo tiempo.

Para más información sobre la microarquitectura Nehalem, se recomienda revisar [16].

Referencias

[1]. Blake, G. Dreslinski, R. Mudge, T.
(Noviembre 2009). A survey of multicore processors.
IEEE Signal Processing Magazine, 26, 26-37.

[2]. Sutter, H. (Marzo 2005). The Free Lunch is Over.
Revisado el 4 de Noviembre de 2010, de http://www.gotw.ca/publications/concurrency-ddj.htm

[3]. Intel Corporation.
Intel's Tick-Tock Model.

Revisado el 4 de Noviembre de 2010, de http://www.intel.com/technology/tick-tock/index.htm

[4]. Hennessy, J.L., Patterson, D.A. (2006). Computer architecture: A quantitative approach. (4a. ed.) EE. UU.: Morgan Kaufmann.

[5]. Hennessy, J.L., Patterson, D.A. (2007). Computer Organization and Design: The Hardware/Software interface. (3a. ed.) EE. UU.: Morgan Kaufmann.

[6]. Blelloch, G. E., Maggs, B. M. Parallel Algorithms.
Revisado el 4 de Noviembre de 2010, de http://www.cs.cmu.edu/afs/cs/academic/class/15499-s09/www/#pa

[7]. Leiserson, C. Demaine, E. 6.046J Introduction to Algorithms (SMA 5503), Fall 2005. Massachusetts Institute of Technology: MIT OpenCourseWare.
Revisado el 4 de Noviembre de 2010, de http://ocw.mit.edu.

[8]. Kim, H. Bond, R.
(Noviembre 2009). Multicore Software Technologies.
IEEE Signal Processing Magazine, 26, 1-10.

[9]. Ayguadé, E. Copty, N. Duran, A. Hoeflinger, J. Lin, Y. Massaioli, F., *et al*.
(Marzo 2009). The Design of OpenMP Tasks.
IEEE Transactions on Parallel and Distributed Systems, 20, 404-418.

[10]. Intel Corporation.
Intel's Cilk Plus.
Revisado el 4 de Noviembre de 2010, de http://software.intel.com/en-us/articles/intel-cilk/

[11]. Leiserson, C. MIT CSAIL y Cilk Arts Inc.
(Julio 2009). The Cilk++ Concurrency Platform.
DAC '09.

[12]. Blumofe, R.D. Joerg, C.F. Kuszumalu, B.C. Leiserson, C. Randall, K.H. Zhou, Y.
(Julio 1995). Cilk: An Efficient Multithreaded Runtime System.
5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[13]. Intel Corporation.
Intel Manycore Testing Lab.
Revisado el 4 de Noviembre de 2010, de http://software.intel.com/en-us/articles/intel-many-core-testing-lab/

[14]. Intel Corporation.
Intel Xeon Processor 7000 sequence
Revisado el 4 de Noviembre de 2010, de
http://www.intel.com/p/en_US/products/server/processor/xeon7000/

[15]. Intel Corporation.
(Enero 2009). An Introduction to the Intel Quickpath Interconnect.
Revisado el 4 de Noviembre de 20010, de
http://www.intel.com/technology/quickpath/introduction.pdf

[16]. Casazza, J. Intel Corporation.
(2009). First the Tick, now the Tock: Intel Microarchitecture (Nehalem).
http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf

Capítulo 3. Algoritmos seleccionados; versiones seriales y paralelización

Es importante que los algoritmos que se seleccionen puedan demostrar los distintos aspectos de la programación paralela.

Para el desarrollo de este trabajo, se han seleccionado los siguientes 4 algoritmos:

1. El cálculo recursivo de los números de Fibonacci
2. El problema de las N Reinas
3. El algoritmo de ordenamiento por mezcla (Mergesort)
4. El algoritmo del vendedor viajero resuelto con búsqueda local (Tabu Search)

Para estos cuatro algoritmos existen diversos aspectos relevantes que se observan al paralelizar un algoritmo. A continuación son presentados los procesos en que cada uno fue programado y paralelizado.

1. El cálculo recursivo de los números de Fibonacci

Versión serial del algoritmo

Es bien sabido que calcular la serie de Fibonacci recursivamente es terriblemente ineficiente. Tiene una complejidad de $O(2^n)$, mientras que el cálculo iterativo tiene complejidad $O(n)$ y es posible alcanzar $O(\log n)$; sin embargo, el algoritmo recursivo tiene mucho paralelismo, y es posible observar el desempeño del sistema de ejecución y el costo de creación de hilos. Recordemos que el objetivo de este trabajo no es desarrollar implementaciones más eficientes de algoritmos existentes, sino comprender el comportamiento y los factores que afectan el desempeño de un algoritmo implementado en un entorno paralelo.

El cálculo de la serie de Fibonacci se realiza mediante la siguiente recursión, que es la definición básica de la serie:

$$fib(n) = fib(n-1) + fib(n-2)$$

Dado que la serie de Fibonacci está definida únicamente para los números naturales, además de esta recursión se definen los valores iniciales de la serie:

$$fib(0) = 0 \; y \; fib(1) = 1$$

Partiendo de estos dos valores iniciales es posible calcular el número de Fibonacci de cualquier número natural mediante la recursión de la función. Cabe señalar que la implementación recursiva de la serie de Fibonacci es ineficiente. El código es el siguiente:

```
fib(n){
      if (n < 2)
            return n;
      else
            return fib(n-1)+fib(n-2);
}
```

Por lo tanto, el árbol de recursión para el cálculo de los números de Fibonacci es:

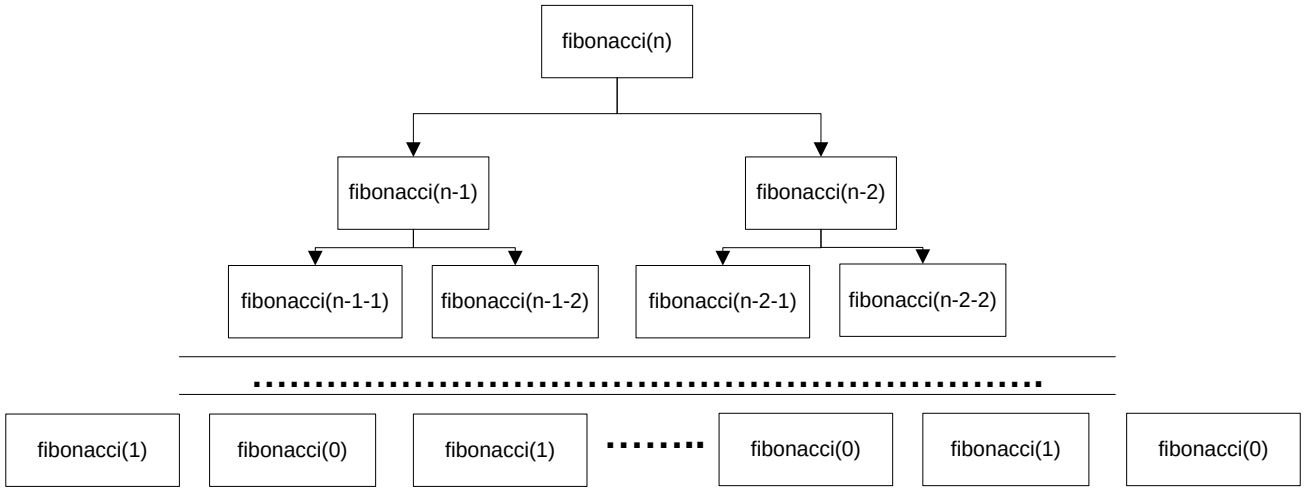

Figura 4. Árbol de recursión de la serie de Fibonacci.

De este árbol es posible comprender la complejidad del algoritmo. De hecho, el cálculo de la complejidad se deriva de la propia recursión que define la función. Dado que cada nivel llama a la función dos veces, en N niveles (desde N hasta 0, disminuyendo de uno en uno), la complejidad del algoritmo es $O(2^n)$.

Paralelización del algoritmo

- Deducción del algoritmo paralelo

Afortunadamente, el cálculo de los números de Fibonacci es un algoritmo con mucho paralelismo. Dado que cada llamada a la función genera dos llamadas recursivas a la misma, es posible ejecutar ambas en paralelo.

A continuación, el código de la versión serial:

```
fib(n){
      if (n < 2)
            return n;
      else
            return fib(n-1)+fib(n-2);
}
```

Dado que ambos cálculos van a ser hechos en paralelo, es necesario separar el cálculo de la salida, para evitar cualquier carrera de datos en la función. Recordemos que una carrera de datos ocurre cuando dos hilos acceden a la misma posición de memoria y al menos uno de ellos la modifica. Para evitar las carreras de datos, es necesario crear una variable independiente en la que cada función devolverá su valor, y luego sumar el resultado de estas dos variables.

La versión paralela del cálculo de los números de Fibonacci es la siguiente:

```
void fibonacci(int * fib, int number){
      if ( number < 2 )
            *fib = number;
      else{
            int a, b;
                  cilk_spawn fibonacci(&a, number-1);
                  cilk_spawn fibonacci(&b, number-2);
                  cilk_sync;
                  *fib = a + b;
      }
}
```

Esta nueva versión del programa está realizada con el lenguaje Cilk++, de Intel. Como se menciona en el capítulo anterior, Cilk++ es un lenguaje que agrega tres palabras reservadas y algunas librerías a C y C++. En este programa se utilizaron las palabras `cilk_spawn` y `cilk_sync`. Estas palabras son utilizadas para expresar paralelismo y para sincronización:

`cilk_spawn` es una palabra reservada del lenguaje Cilk++, y permite 'crear' una tarea paralela que puede ser ejecutada al mismo tiempo que el hilo que la lanza. Cabe aclarar que `cilk_spawn` expresa paralelismo, no comanda la ejecución de la tarea. Dado que Cilk++ provee una capa de software que se encarga de la planificación de las tareas, el programador sólo *expresa* el paralelismo, y Cilk++ se encarga de planificar su ejecución.

En este caso, la línea `cilk_spawn fibonacci(&a, number-1)` crea una tarea que puede ser ejecutada en paralelo, y la línea `cilk_spawn fibonacci(&b, number-2)` crea otra tarea. Esto significa que estas dos tareas pueden ser ejecutadas en paralelo.

`cilk_sync` es otra palabra reservada de Cilk++, y se utiliza para realizar la sincronización de las tareas y los hilos de ejecución. Esta palabra crea una *barrera* que impide que se avance en la ejecución del programa hasta que todas las tareas creadas en su contexto hayan sido completamente ejecutadas.

En este caso, la línea `cilk_sync;` impide que se avance en la ejecución del programa hasta que ambas llamadas recursivas en paralelo hayan sido ejecutadas. Esto es importante porque si se continúa la ejecución antes de asegurar que ambas tareas hayan sido ejecutadas, la línea `*fib = a + b;` tendría resultados no determinísticos, ya que no es posible saber en qué punto de la ejecución de las tareas paralelas se está.

Para la versión paralela del programa, la declaración de la función es diferente: Ahora necesitamos recibir como argumento la variable en donde se va a guardar el resultado, para evitar las carreras de datos, y que cada función trabaje con un set de variables independientes. Es interesante ver que esto surge de la necesidad de crear paralelismo de datos donde no lo hay. Afortunadamente no es un problema demasiado complejo en este caso, y sólo es necesario asignar algunas variables extra para mantener la escalabilidad del código.

- La profundidad y el trabajo

La profundidad de este nuevo algoritmo se calcula con base a la cantidad de operaciones que realiza cada llamada:

$$D(n) = \max\big(D(n-1), D(n-2)\big) + 2$$

Luego, se deriva lo siguiente:

$$D(n) = D(n-1) + O(1)$$

Esta recursión es muy sencilla de resolver; la profundidad de este algoritmo es entonces:

$$D(n) = O(n)$$

El trabajo puede ser obtenido igual que la complejidad de tiempo. De hecho, la recursión y la derivación es idéntica y el trabajo es:

$$W(n) = O(2^n)$$

Sabiendo esto es posible calcular el paralelismo, que simplemente se obtiene de dividir el trabajo entre la profundidad. Dado que todo esto está en función de la entrada el resultado también está en función de la entrada y es el siguiente:

$$P(n) = O\left(\frac{2^n}{n}\right)$$

- Análisis de la implementación paralela

La implementación paralela del algoritmo para calcular los números de Fibonacci tiene trabajo del orden de la complejidad temporal de la implementación serial, es decir que $T_{serial}\ (n) = \theta(W(n))$. Cuando un algoritmo paralelo cumple con esto, se dice que el algoritmo es **efectivo**. Es deseable que un algoritmo paralelo realice asintóticamente igual trabajo que su versión serial.

Como se puede ver, dado que el paralelismo depende de la entrada, cuando la entrada es suficientemente grande, el paralelismo se dispara. Es decir que para una entrada suficientemente grande, **el tiempo de ejecución del algoritmo debe disminuir linealmente en función del número de procesadores en que el programa sea ejecutado**, para un número de procesadores suficientemente inferior al paralelismo del programa.

También, dado que las tareas son tan cortas, al probar la ejecución del programa, será posible observar el desempeño y el costo de creación y planificación de tareas en un solo núcleo respecto de la versión serial del algoritmo. Es decir, será posible medir cómo afectan los costos de paralelización a la efectividad del algoritmo.

2. El problema de las N reinas

Versión serial del algoritmo

El problema de las N reinas se trata de un acertijo en el que es necesario colocar N reinas en un tablero de ajedrez de NxN de tal manera que ninguna de las reinas esté amenazando a otra. Este problema es utilizado a varios niveles de enseñanza, y es excelente en el estudio de algoritmos.

Este problema puede ser resuelto por varios métodos: búsqueda de fuerza bruta, algoritmos genéticos, mejora iterativa, etc. En este trabajo se utiliza un algoritmo de búsqueda de profundidad con *backtracking* que permite disminuir la cantidad de estados a revisar respecto de la búsqueda por fuerza bruta. El tiempo de ejecución tomado en cuenta será el tiempo que tome obtener todas las soluciones.

El algoritmo consiste en dividir el tablero en N columnas distintas, y empezar a colocar las reinas en cada columna. Primero, colocar una reina en la primera fila disponible de la columna actual (donde una fila disponible es aquella que no esté siendo atacada) y pasar a la siguiente columna. Si la columna que se está revisando ya no tiene filas disponibles, es necesario volver y mover la reina en la columna anterior a la siguiente fila disponible de dicha columna. Una vez que se encuentra una solución, se agrega a la cuenta, y se continúa el recorrido del árbol.

Para simplificar el análisis del problema, primero se ha realizado en un tablero de 4x4, y de ahí es posible generalizar para un tablero de cualquier tamaño:

0,0	0, 1	0,2	0,3
1, 0	1, 1	1, 2	1,3
2, 0	2, 1	2, 2	2,3
3, 0	3, 1	3,2	3,3

Figura 5. Tablero de ajedrez de 4x4.

Para modelar las diagonales del tablero es de gran ayuda observar que en aquellas diagonales ascendentes de izquierda a derecha, la suma de los índices de fila y columna es la misma, mientras que en aquellas descendentes de izquierda a derecha, la diferencia de los índices de fila y columna es siempre la misma. Esto nos permite direccionar un arreglo de diagonales izquierdas (con la suma) y diagonales derechas (con la diferencia) y simplificar la búsqueda de cuadros disponibles.

Este algoritmo puede ser más fácilmente definido con recursividad sobre cada una de las columnas; con un algoritmo desarrollado con orientación a objetos. Mediante un objeto que represente el tablero de ajedrez, y con éste marcar las columnas, filas y diagonales que se encuentren ocupadas.[6]

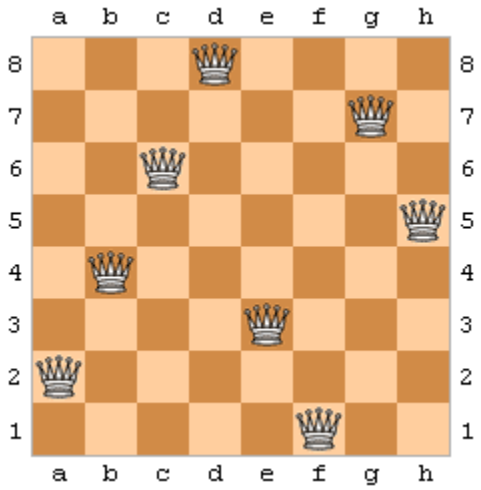

Figura 6. Una solución al problema de las 8 reinas.

La declaración de la clase ChessBoard en C++ es la siguiente:

```
class ChessBoard{
public:
	ChessBoard(int);
	void findSolutions();
	int getSolutions();

private:
	const bool available;
	const int squares, norm;
	bool *column, *leftDiagonal, *rightDiagonal;
	int *positionInRow;
	void putQueen(int);
	void initializeBoard();
	int solutions;
};
```

Hablemos de las variables. Los miembros `column`, `leftDiagonal` y `rightDiagonal` son apuntadores a arreglos que guardan la información sobre si una columna, diagonal izquierda o diagonal derecha están ocupados. El miembro `positionInRow` es un apuntador a un arreglo que contiene la columna en que fue colocada la reina de cada fila. El miembro `solutions` es una variable entera que guarda la cantidad de soluciones que el programa ha encontrado. Finalmente, los miembros `available`, `squares` y `norm` son constantes que guardan respectivamente el valor `true`, la

[6] La versión serial del algoritmo fue tomada de [1].

cantidad de cuadros que hay en cada lado del tablero y el desplazamiento necesario para normalizar el direccionamiento de las diagonales del tablero.

Las funciones miembro de la clase permiten inicializar todos los arreglos y buscar la solución. A continuación se presenta el constructor y la función de inicialización de la clase, que únicamente se encargan de crear los arreglos y limpiar la información que contienen:

```
ChessBoard::ChessBoard(int n):available(true), squares(n), norm(squares-1)
{
      initializeBoard();
}

void ChessBoard::initializeBoard(){
      register int i;
      column = new bool[squares];
      positionInRow = new int[squares];
      leftDiagonal = new bool[squares*2-1];
      rightDiagonal = new bool[squares*2-1];

      for(i=0; i<squares; i++){
            positionInRow[i]=-1;
            column[i] = available;
      }
      for(i=0; i<squares*2-1; i++){
            leftDiagonal[i] = rightDiagonal[i] = available;
      }
      solutions = 0;
}
```

Finalmente, el núcleo del programa es la función de búsqueda, que se implementa de manera recursiva sobre cada una de las filas del tablero. Vale la pena estudiar el algoritmo cuidadosamente, para revisar cómo se modela el problema en arreglos y cómo se utilizan para ir colocando las reinas. Como se puede ver, es una búsqueda a profundidad con *backtracking*:

```
void ChessBoard::putQueen(int row){
            for(int col = 0; col < squares; col++)
            if (column[col] == available &&
                  leftDiagonal[row+col] == available &&
                  rightDiagonal[row-col+norm] == available){
                        positionInRow[row] = col;
                        column[col] = !available;
                        leftDiagonal[row+col] = !available;
                        rightDiagonal[row-col+norm] = !available;
                        if ( row < squares-1 )
                              putQueen(row+1);
                        else solutions = solutions + 1; //una solucion
                        column[col] = available;
                        leftDiagonal[row+col] = available;
                        rightDiagonal[row-col+norm] = available;

            }
}
```

- Complejidad de tiempo del programa serial de las N reinas

La complejidad de tiempo del núcleo del programa serial de las N reinas puede ser calculada de manera muy sencilla partiendo del código. El algoritmo cuenta con un ciclo `for`, dentro de este ciclo hay algunas operaciones de complejidad $O(1)$ y una recursión. Es importante observar que la recursión no disminuye en complejidad, ya que el tablero sigue teniendo n posiciones en cada fila sobre la que se realiza la recursión.

Para derivar la expresión general de la complejidad, es necesario primero contar con el caso base, es decir, la complejidad en la última recursión de la función:

$$T_{n-1}(n) = O(n)$$

Ahora derivamos la complejidad de las primeras recursiones; dado que la función cuenta con algunas operaciones constantes y una recursión, la ecuación es la siguiente:

$$T_i(n) = O(n) + nT_{i+1}(n), \text{ donde } i\epsilon[0 \dots n-1]$$

Del caso base y la recursión, se obtiene la complejidad de tiempo del algoritmo, de la siguiente manera:

$$T_0(n) = O(n) + nT_1(n) = O(n) + nO(n) + n^2T_2(n)$$

Luego entonces:

$$T_0(n) = O(n) + nT_1(n) = O(n) + O(n^2) + \cdots + O(n^n)$$

De donde se deriva que la complejidad temporal del algoritmo serial de las N Reinas es:

$$T_0(n) = O(n^n)$$

Paralelización del algoritmo

- La deducción del algoritmo paralelo y las nuevas estructuras de datos

Obtener una versión paralela del algoritmo que calcula los números de Fibonacci fue relativamente sencillo. Sólo fue necesario evitar las carreras de datos en las dos llamadas recursivas. Sin embargo, el problema de las N reinas es mucho más complicado de paralelizar.

Sabemos que este problema es una búsqueda a profundidad sobre un árbol con profundidad N y factor de ramificación de N. Afortunadamente este algoritmo tiene mucho paralelismo implícito, dado que cada rama del árbol puede ser recorrida independientemente. En la figura 7 se muestra de manera simple el árbol de recursión de la aplicación del algoritmo sobre un tablero de 4x4 cuadros.

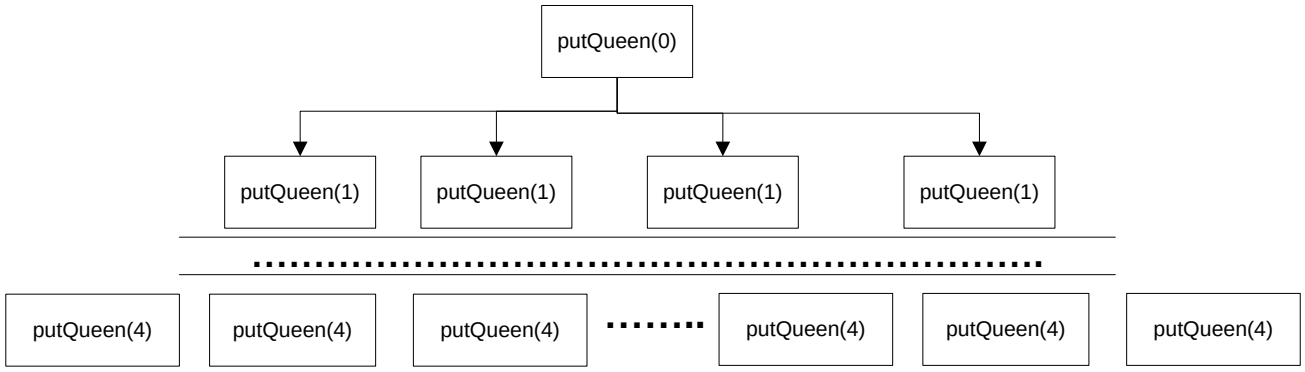

Figura 7. Árbol de recursión del problema de las 4 reinas.

Es fácil ver que cada rama del árbol es completamente independiente, ya que todas expresan posiciones diferentes de las reinas en el tablero. Es decir que el algoritmo tiene un grado muy alto de paralelismo.

Existe un detalle que complica la paralelización: el programa serial para resolver el problema de las N reinas utiliza tres arreglos para marcar las casillas del tablero que están disponibles. En estos tres arreglos, el algoritmo guarda las filas, diagonales izquierdas y diagonales derechas que están siendo amenazadas por una reina colocada anteriormente. Dado que el algoritmo se ejecuta de manera serial, en cualquier momento de la ejecución, sólo es necesario guardar una copia de estos arreglos, por que el algoritmo únicamente tiene en cuenta una configuración. Sin embargo, al ejecutar el algoritmo en paralelo, existirán varias configuraciones del tablero en un mismo punto en el tiempo. Esto significa que las estructuras de datos utilizadas en el algoritmo serial no son aptas para la versión paralela, y es necesario diseñar una nueva estructura que sea *ad hoc* para las necesidades del algoritmo.

En este trabajo se revisan dos estrategias para paralelizar las estructuras de datos. La primera surgió de analizar el algoritmo y sus cualidades y necesidades. La segunda surgió como una estrategia para mejorar el desempeño de la primera, que resultó complicar demasiado el algoritmo. A continuación se describe la derivación de ambas estrategias.

Para diseñar la estructura de datos, es necesario tener en cuenta ciertas especificaciones necesarias. El algoritmo recursivo permite aplicar la estrategia de divide y conquista, así que cada vez que la función es

llamada recursivamente, debe poder acceder a las posiciones de las reinas colocadas anteriormente; esto manteniendo la integridad de los datos de cada rama del árbol. La estructura en forma de árbol del algoritmo sugiere una estrategia para codificar el problema: una estructura de datos en forma de árbol podría ayudar a modelar las posiciones de las reinas. Estas llamadas de función van construyendo un árbol; pero dado que cada función necesita conocer la lista con las posiciones de las reinas colocadas anteriormente, este árbol tiene una forma inversa, donde los hijos apuntan hacia sus padres, y así cada llamada de función en paralelo tiene acceso únicamente a los nodos del árbol que están en su trayectoria.

La estructura de datos que representa las listas de reinas colocadas es la siguiente:

```
class NodePosition{
public:
      NodePosition(int, int);
      NodePosition(int, int, NodePosition*);
      bool available(int, int);
      NodePosition* insert(int, int);
      int getColumn();
      int getRow();
private:
      int column;
      int row;
      NodePosition *previous;
};
```

Una instancia de esta clase guarda la posición de una reina en sus miembros `column` y `row`; además de la reina colocada en la columna anterior. Los miembros `available(int, int)` e `insert(int, int)` realizan, respectivamente, la comprobación de que cierta posición esté disponible en la lista, o insertan un nodo nuevo a la lista, con una reina en la posición pasada en los argumentos. A continuación se muestra el código de ambos miembros:

```
bool NodePosition::available(int test_column, int test_row){
      if ( row == test_row ||
      //Verificar si la fila esta ocupada
            column == test_column ||
      //Verificar si la columna esta ocupada
            (row + column) == (test_row + test_column) ||
      //Verificar si la diagonal izquierda esta ocupada
            (row - column) == (test_row - test_column) )
      //Verificar si la diagonal derecha esta ocupada
                  return false;
      //En este caso, el cuadro no esta disponible

      else if ( previous == 0 )
            return true;
      else
            return previous->available(test_column, test_row);
}

NodePosition* NodePosition::insert(int insert_column, int insert_row){
      return new NodePosition(insert_column, insert_row, this);
}
```

Esta estructura de datos parece más bien como una lista simplemente ligada; sin embargo, vale la pena revisar el resto del código de la aplicación para entender porqué al ejecutarse, se crean árboles 'invertidos', es decir, donde los hijos apuntan a los padres.

La clase `ChessBoard` es muy similar a la clase utilizada en el algoritmo paralelo, con la diferencia de que la versión paralela utiliza la clase `NodePosition` para saber el estado actual del tablero, en vez de arreglos miembros de la propia clase. La declaración de la clase `ChessBoard` es la siguiente:

```
class ChessBoard{
public:
      ChessBoard();
      ChessBoard(int);
      void findSolutions();
      int getSolutions();
      int getSquares();
private:
      const int squares;
      int solutions;
      int putQueen(NodePosition*, int, int*);
};
```

En este caso, la función `putQueen` toma como parámetro la lista de reinas colocadas anteriormente, para poder decidir si colocar o no una reina en cada espacio de la fila correspondiente. El resto de los elementos son obviados por su sencillez. A continuación se presenta el código del algoritmo para realizar la búsqueda en paralelo:

```
void ChessBoard::putQueen(NodePosition *position, int row, int *saveat){
      if ( row == squares ){
            *saveat += 1;
            delete position;
            return;
      }

      int *solutions = new int;
            *solutions = 0;
      if ( position == 0 ){
            for (int col = 0; col < squares; col++){
      cilk_spawn putQueen(new NodePosition(col, row), row+1, solutions);
            }
      }
      else{
            for (int col = 0; col < squares; col++){
                  if ( position->available(col, row) ){
      cilk_spawn putQueen( position->insert(col, row), row+1, solutions);
                  }
            }
      }
      cilk_sync;

      *saveat += *solutions;
      delete solutions;
      delete position;
}
```

De esta manera, el algoritmo crea tareas nuevas por cada rama del árbol de recursión, pero además crea un nuevo nodo de la lista en cada rama. Antes de pasar a describir la forma en que el árbol de configuraciones del tablero sería creado, es importante mencionar que este algoritmo está incompleto: aunque el código parece correcto, existe una condición de carrera. Esta condición de carrera ocurre porque todos los hilos escriben sobre la misma variable, `solutions`, y esto puede crear inconsistencias de datos al estar operando sobre la misma dirección de memoria desde varios procesadores. Para evitar las carreras de datos es necesario crear datos independientes en los que cada hilo pueda escribir, para sumar los resultados posteriormente. En este caso, el nuevo algoritmo que soluciona las carreras de datos existentes en el anterior es el siguiente:

```
int ChessBoard::putQueen(NodePosition *position, int row){
      if ( row == squares ){
            delete position;
            return 1;
      }

      int *solutions = new int[squares];
      for (int i=0; i<squares; i++)
            *(solutions + i) = 0;
      if ( position == 0 ){
            for (int col = 0; col < squares; col++){
solutions[col] = cilk_spawn putQueen( position->insert(col, row), row+1);
            }
      }
      else{
            for (int col = 0; col < squares; col++){
                  if ( position->available(col, row) ){
solutions[col] = cilk_spawn putQueen( position->insert(col, row), row+1);
                  }
            }
      }
      cilk_sync;
      int value = 0;
      for( int i = 0; i < squares; i++ ){
            value = value + solutions[i];
      }
      delete solutions;
      delete position;
      return value;
}
```

En este caso, el paralelismo se explota 'colocando reinas en distintos tableros nuevos', y haciéndolo en paralelo. Las líneas `solutions[col] = cilk_spawn putQueen( position->insert(col, row), row+1);` crean las tareas paralelas colocando reinas en distintos cuadros para seguir avanzando. La cantidad de soluciones que una función encuentra es guardada en el arreglo `solutions` y las soluciones son reducidas posteriormente.

Por esto es necesario agregar la línea `cilk_sync;` justo después del ciclo donde se crean las tareas y antes del ciclo de reducción. Esto permite asegurar resultados determinísticos en el programa y que no existan carreras.

Se crea un arreglo con N elementos, para así particionar el espacio en el que cada hilo va a trabajar, y evitar las carreras de datos. Es interesante observar que aunque esto produce un programa de comportamiento determinístico, también crea un problema de desempeño: la suma de las soluciones de los hilos hijos se realiza de manera serial en cada hilo padre, y esto puede limitar el desempeño del algoritmo.

A continuación, a manera de ejemplo, se muestra el árbol inverso de configuraciones del tablero que se crea con una búsqueda sobre un tablero de 4x4 cuadros:

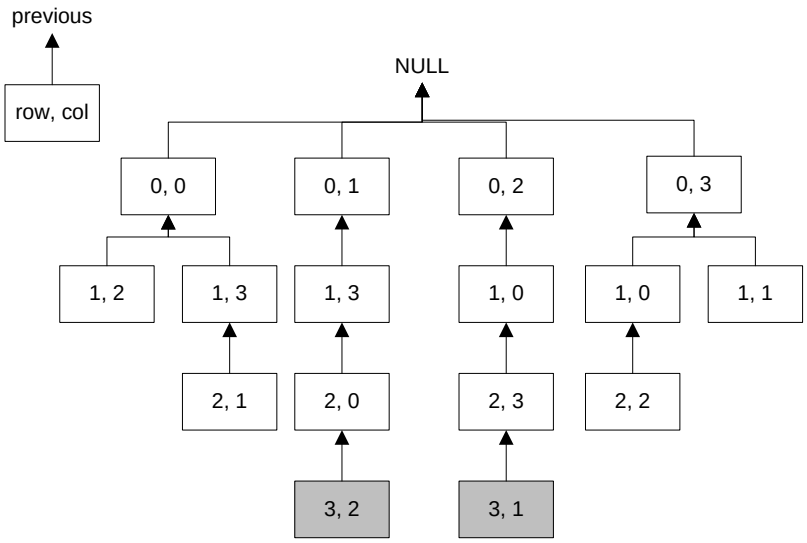

Figura 8. Árbol inverso de la resolución del problema de las 4 reinas. Los nodos grises son nodos de soluciones finales.

- Cálculo del trabajo y la profundidad

Vale dividir el código en segmentos y calcular la complejidad de cada segmento por separado:

```
        if ( row == squares ){
              delete position;
              return 1;
        }

        int *solutions = new int[squares];

        […]

        delete solutions;
        delete position;
        return value;
}
```

La primera sección del código, que consiste en el caso base y la creación de las variables es de complejidad constante; tanto su profundidad como su trabajo son $O(1)$.

La segunda sección del código consiste en dos ciclos `for` en secuencia. Hay dos casos en los que cada una de las secciones se ejecuta, uno es el caso inicial, donde la lista de configuración del tablero no está inicializada y el otro es el caso donde la lista de configuración ya tiene al menos un elemento. Dado que en general, el caso más común será el caso donde la lista ya contenga algún elemento, y para poder estimar el efecto de la búsqueda en el algoritmo, se revisará este caso como el caso general:

```
      for (int i=0; i<squares; i++)
            *(solutions + i) = 0;

            for (int col = 0; col < squares; col++){
                  if ( position->available(col, row) ){
solutions[col] = cilk_spawn putQueen( position->insert(col, row), row+1);
                  }
            }
      cilk_sync;
      int value = 0;
      for( int i = 0; i < squares; i++ ){
            value = value + solutions[i];
      }
```

El primer ciclo `for` es un ciclo secuencial para inicializar el arreglo de soluciones de tamaño n, donde n es la cantidad del cuadros del tablero, no la fila en que se coloca la reina. El trabajo y la profundidad del primer ciclo `for` son ambos $O(n)$. El segundo ciclo `for` es necesario analizarlo de adentro hacia afuera.

En la línea `solutions[col] = cilk_spawn putQueen( position->insert(col, row), row+1);` se tiene una recursión. Esta recursión es sobre una función de n, no de n-1.

La búsqueda sobre el tablero que se realiza en `position->available(col, row)` es de orden n. Aunque normalmente la búsqueda se realiza en menos de n elementos de la lista, estos crecen proporcionalmente con el tamaño del tablero, así que la búsqueda es $O(n)$ en trabajo y profundidad.

Finalmente, el segundo ciclo `for`, donde se reducen los valores acumulados en cada uno de los elementos del arreglo de soluciones, es $O(n)$.

De aquí es posible derivar tanto el trabajo como la profundidad de la recursión. Ya con la complejidad de todos los segmentos de código calculada, obtenemos que:

$$D_i(n) = O(n) + nO(n) + D_{i+1}(n) = O(n^2) + D_{i+1}(n)$$

y $W_i(n) = O(n) + nO(n) + nW_{i+1}(n) = O(n^2) + nW_{i+1}(n)$, donde $i \epsilon [0..n-1]$

Estas recursiones no expresan la complejidad del caso base. El caso base tiene complejidad constante, ya que sólo se ejecuta el primer segmento de código de la recursión; es decir que:

$$D_n(n) = O(1) \text{ y } W_n(n) = O(1)$$

Con las recursiones y el caso base es posible derivar la expresión de la complejidad sin recursión. Revisando el árbol de recursión de la figura 8 es posible obtener cierta intuición sobre la forma general de la profundidad y el trabajo.

La forma general de la profundidad es la siguiente:

$$D_0(n) = O(n^2) + D_1(n) = O(n^2) + O(n^2) + D_2(n) = nO(n^2) + D_n(n) = nO(n^2) + O(1)$$

De donde se deriva que:

$$D(n) = O(n^3)$$

Es decir que la profundidad es de orden n cúbica.

La forma general del trabajo es un tanto más difícil de derivar. A continuación es la derivación de la forma general del trabajo:

$$W_0(n) = O(n^2) + nW_1(n) = O(n^2) + n\big(O(n^2) + nW_2(n)\big) = O(n^2) + O(n^3) + n^2W_2(n)$$

De esto, es posible derivar lo siguiente:

$$W_0(n) = O(n^2) + O(n^3) + \cdots + O(n^{n+1})$$

Es decir, que la expresión general del trabajo es:

$$W(n) = O(n^{n+1})$$

Una vez que conocemos el trabajo y la profundidad del algoritmo, podemos obtener el paralelismo, que se obtiene de dividir estos dos:

$$P(n) = O\left(\frac{n^{n+1}}{n^3}\right) = O(n^{n-2})$$

- Análisis teórico de la implementación paralela

Como vemos, el algoritmo paralelo, a comparación de la versión serial NO es efectivo. De hecho, su trabajo es de orden n veces mayor que su complejidad temporal, como vemos, $W(n) = O(n^{n+1})$ y $T(n) = O(n^n)$, así que $\frac{W(n)}{T(n)} = O(n)$. Esto significa que la ejecución del algoritmo paralelo en un solo núcleo será mucho más lenta que la ejecución del algoritmo serial.

Por otro lado, el paralelismo del algoritmo es mucho muy alto, esto permite que el algoritmo sea muy escalable paralelamente; esto significa que al aumentar el número de núcleos disponibles en el sistema, el tiempo de ejecución mejorará considerablemente. Es decir que existe una cantidad n donde si el programa paralelo es ejecutado en n o más núcleos, su tiempo de ejecución será menor que el del algoritmo serial.

Desafortunadamente, el trabajo que realiza el algoritmo es demasiado, y esto detiene su desempeño. Vale la pena realizar reingeniería del algoritmo paralelo, para poder acelerar su ejecución y mejorar su efectividad.

Reingeniería del algoritmo paralelo

El problema con el primer algoritmo paralelo para resolver el problema de las N reinas es que no es efectivo; de hecho, su trabajo es de orden n^{n+1} y la complejidad temporal del algoritmo serial es de

orden n^n. Para mejorar el algoritmo es necesario encontrar la sección del algoritmo de donde sale ese orden n extra que vuelve al programa inefectivo, y rediseñarla de manera que esto no ocurra.

Las derivaciones del trabajo y la complejidad temporal del algoritmo, respectivamente, son las siguientes:

$$W_i(n) = O(n) + n\big(O(n) + W_{i+1}(n)\big) = O(n^2) + nW_{i+1}(n)$$

$$T_i(n) = O(n) + nT_{i+1}(n)$$

Como vemos, las derivaciones son distintas. Existe un término más en el trabajo, que no se tiene en la complejidad temporal. Este sumando extra crea un término de orden n cuadrada en la derivación, y este exponente extra es el que crea el orden n obtenido en la división $\frac{W(n)}{T(n)} = O(n)$.

¿De dónde viene este término extra?

```
for (int col = 0; col < squares; col++){
      if ( position->available(col, row) ){
```

De hecho, este término extra viene de la búsqueda, ya que ésta es $O(n)$, y es realizada n veces en la ejecución del ciclo principal, esto nos crea el término $O(n^2)$; además es ejecutada de manera serial, así que no sólo empeora el trabajo, sino también la profundidad, y el paralelismo. Es decir que la búsqueda está afectando el desempeño del algoritmo en todos los niveles. Es necesario acelerar la búsqueda.

En el algoritmo serial, la búsqueda se realiza con otro esquema:

```
if (column[col] == available &&
      leftDiagonal[row+col] == available &&
      rightDiagonal[row-col+norm] == available){
```

En este caso, ni siquiera hay llamadas a otras funciones: la búsqueda se realiza al revisar la disponibilidad de las diagonales y las columnas del tablero, con trabajo de orden constante $O(1)$.

El bajo orden de complejidad de la búsqueda en el algoritmo serial se debe a que los arreglos miembro de la clase `ChessBoard` permiten realizarla directamente, mientras que la búsqueda dentro de la lista se realiza atravesando la lista elemento por elemento.

La búsqueda en la lista no puede ser acelerada mucho más. Esto significa que es necesario cambiar la estructura de datos donde se guarda la configuración del tablero para poder mejorar el orden del trabajo del algoritmo.

Otra sección del algoritmo donde se daña el desempeño del programa es la reducción. La reducción es el agregado de los resultados de todas las tareas en una sola variable para evitar carreras de datos y que exista unidad de información; y es de hecho un problema común en el cómputo paralelo. En muchas ocasiones es necesario agregar información calculada en paralelo, y la reducción es el paso en el que se

realiza. En el programa para el cálculo de los números de Fibonacci existe también un paso de reducción, pero éste es un tanto más sencillo, ya que sólo existen dos tareas hijas por cada padre. En el caso del problema de las N reinas, la reducción es costosa ya que tiene trabajo y profundidad $O(n)$. En el siguiente segmento de código se muestra la sección de reducción:

```
for( int i = 0; i < squares; i++ ){
      value = value + solutions[i];
}
```

Por otro lado, es importante tomar en cuenta el gasto en tiempo de la intercomunicación entre procesadores y la creación de tareas a las que es necesario pasar demasiados argumentos, así que pasar los arreglos de configuración completos tampoco es la opción adecuada.

Los arreglos de configuración de las columnas y las diagonales son arreglos booleanos, es decir que cada elemento guarda únicamente un bit, que informa si la columna o la diagonal está disponible. Esto sugiere una opción para comprimir la información: utilizar los bits dentro de las palabras; es decir, utilizar las variables enteras tipo `int`, o `long` como arreglos booleanos compactos, donde cada elemento es de un solo bit, y así es posible pasar la configuración del tablero como argumento a la función recursiva en una sola palabra de memoria, sin crear demasiado gasto por la creación de tareas paralelas con demasiados argumentos.

Esto revela que es necesario realizar reingeniería también en la clase `ChessBoard` para adaptarse a las necesidades de la nueva implementación. A continuación se presenta la estructura de la nueva clase, en la que se incluye un arreglo `positions` para comprobar la disponibilidad de las posiciones en las palabras de configuración del tablero; además se incluye un objeto de tipo `cilk::reducer_opadd<int>`; esta clase se encuentra incluida en el framework de Cilk++, y permite reducir los resultados de diversas tareas que pueden ser ejecutadas en paralelo en una sola variable, sin provocar carreras de datos[7]. También se muestra la inicialización del 'arreglo' de posiciones, que no es más que un arreglo donde cada elemento cuenta un bit marcado en la posición especificada.

[7] Para más información respecto de las carreras de datos y los reductores, revisar [2].

```
class ChessBoard{
public:
      ChessBoard();
      ChessBoard(int);
      void findSolutions();
      int getSolutions();
      int getSquares();
private:
      cilk::reducer_opadd<int> total;
      const int squares;
      const int norm;
      int *positions;
      int solutions;
      void putQueen(int, long, long, long);
      void initializeChessBoard();
};

void ChessBoard::initializeChessBoard(){
      solutions = 0;
      positions = new int[2*squares - 1];
      for ( int i = 0, a = 1; i < 2*squares-1; a *= 2, i++ ){
            positions[i] = a;
      }
}
```

Para entender mejor cómo se utiliza el arreglo de posiciones para realizar la búsqueda de posiciones disponibles en el tablero es mejor revisar directamente el código del nuevo algoritmo:

```
void ChessBoard::findSolutions(){
      putQueen(0, 0, 0, 0);
      solutions = total.get_value();
}

void ChessBoard::putQueen(int row, long cols, long leftDiagonals, long
rightDiagonals){
      for (int col = 0; col < squares; col ++){
            if ( !((cols & positions[col]) ||
                  (leftDiagonals & positions[col+row]) ||
                  (rightDiagonals & positions[norm+row-col])) ){
                        if ( row == squares-1){
                              total += 1;
                              break;
                        }
                        else
                        cilk_spawn putQueen(row+1, cols | positions[col],
                  leftDiagonals | positions[col+row], rightDiagonals |
                  positions[norm+row-col]);
            }
      }
      cilk_sync;
}
```

Como vemos, incluso la cantidad de código necesario disminuyó, gracias a que se incluyó un reductor como parte de la clase, y a que la búsqueda se volvió más directa.

Es interesante detenerse en la llamada recursiva a la función, para entender de qué manera se marcan las columnas y las diagonales como ocupadas, así como la forma en que se mantiene la independencia de los datos entre cada tarea paralela:

```
putQueen (int row, long cols, long leftDiagonals, long rightDiagonals)

putQueen(row+1, cols | positions[col], leftDiagonals | positions[col+row],
rightDiagonals | positions[norm+row-col]);
```

El primer argumento, `row`, es el que guarda la fila en que se pretende colocar una nueva reina. Esta fila, junto con el barrido de cada columna permite calcular las direcciones de las diagonales.

Los argumentos `cols`, `leftDiagonals` y `rightDiagonals` son las palabras de configuración; en éstas se guarda un 1 por cada diagonal o columna ocupada, y es por esto que al llamar la función recursivamente, estos argumentos son conjuntados con la posición en que la reina es colocada mediante un `or` lógico.

Por ejemplo, la llamada inicial, `putQueen(0, 0, 0, 0);`, consiste en colocar una reina en la fila 0, cuando ninguna de las columnas o diagonales está ocupada.

La desventaja más evidente de este nuevo algoritmo es el gasto incurrido al pasar argumentos extra a la tarea creada para ser ejecutada en paralelo; sin embargo, este gasto incurrido es $O(1)$, al igual que el nuevo algoritmo de búsqueda, y la reducción. Es decir que la complejidad del nuevo algoritmo se calcula sin el término $O(n^2)$ creado por la búsqueda, y el término $O(n)$ creado por la reducción.

- Trabajo y profundidad del nuevo algoritmo

Dado que el nuevo código es más sencillo, calcular la profundidad y el trabajo también es más sencillo. La única sección del código que se toma para el análisis es la siguiente:

```
      for (int col = 0; col < squares; col ++){
            if ( !((cols & positions[col]) ||
                  (leftDiagonals & positions[col+row]) ||
                  (rightDiagonals & positions[norm+row-col])) ){
                        if ( row == squares-1){
                              total += 1;
                              break;
                        }
                        else
                        cilk_spawn putQueen(row+1, cols | positions[col],
                  leftDiagonals | positions[col+row], rightDiagonals |
                  positions[norm+row-col]);
            }
      }
      cilk_sync;
```

Como ya se había comentado, la búsqueda en la sentencia condicional es $O(1)$. El ciclo `for` tiene profundidad y trabajo de $O(n)$. En el caso base de la recursión, la profundidad y el trabajo vienen en ambos casos del ciclo `for`, es decir que la complejidad del caso base es la siguiente:

$$D_n(n) = O(n)$$

La recursión para la profundidad de este algoritmo es:

$$D_i(n) = O(n) + O(n) + D_{i+1}(n) = O(n) + D_{i+1}(n) \text{ para } i \in [0 \dots n-1]$$

Luego, resolviendo esta recursión:

$$D_0(n) = O(n) + D_1(n) = (n-1)O(n) + D_n(n)$$

Es decir:

$$D_0(n) = O(n^2)$$

Como vemos, la profundidad es de un orden de potencia menor que la profundidad del primer algoritmo. Esto es excelente para el paralelismo de la aplicación y el tiempo de ejecución, que seguramente serán visiblemente inferiores.

La recursión para el trabajo de este algoritmo es:

$$W_i(n) = O(n) + O(n) + nW_{i+1}(n) = O(n) + nW_{i+1}(n) \text{ para } i \in [0 \dots n-1]$$

Luego entonces:

$$W_i(n) = O(n) + n\big(O(n) + W_{i+2}(n)\big)$$

De continuar las derivaciones, el caso resultante es:

$$W_0(n) = O(n^n)$$

El resultado obtenido, como esperado, al igual que la profundidad, es un orden de potencia menor que el primer algoritmo paralelo diseñado. Esto significa que se ha mejorado la efectividad del algoritmo; de hecho, el nuevo algoritmo tiene efectividad $\frac{W(n)}{T(n)} = O(1)$, es decir que el trabajo y el tiempo de ejecución del algoritmo serial son asintóticamente iguales.

Por otro lado, otra ventaja de este algoritmo es que conserva el mismo paralelismo que el primer algoritmo paralelo para resolver el problema de las N reinas, dado que al realizar la división de su trabajo entre su profundidad, el resultado es el mismo:

$$P(n) = O\left(\frac{n^n}{n^2}\right) = O(n^{n-2})$$

Es decir que el algoritmo aún tiene mucho paralelismo, y es escalable, así que según el análisis teórico, la ecuación de la tasa de desempeño lineal se mantiene para un gran número de núcleos de ejecución. Queda ver los resultados al ejecutarse en varios núcleos.

3. Ordenamiento por mezcla

Versión serial del algoritmo

El algoritmo de ordenamiento por mezcla, o *merge sort,* es un algoritmo avanzado de ordenamiento de secuencias. Es un algoritmo que fue diseñado utilizando la técnica de divide y conquista; esto quiere decir que incluso la implementación serial tiene bastante paralelismo implícito.

El algoritmo consiste en ordenar las mitades inferior y superior de la secuencia, y posteriormente mezclar las dos mitades en orden. Este algoritmo funciona recursivamente, hasta que el tamaño de la secuencia a ordenar se vuelve uno. A continuación se muestra un árbol de ordenamientos con el que se ordena una secuencia utilizando *merge sort*:

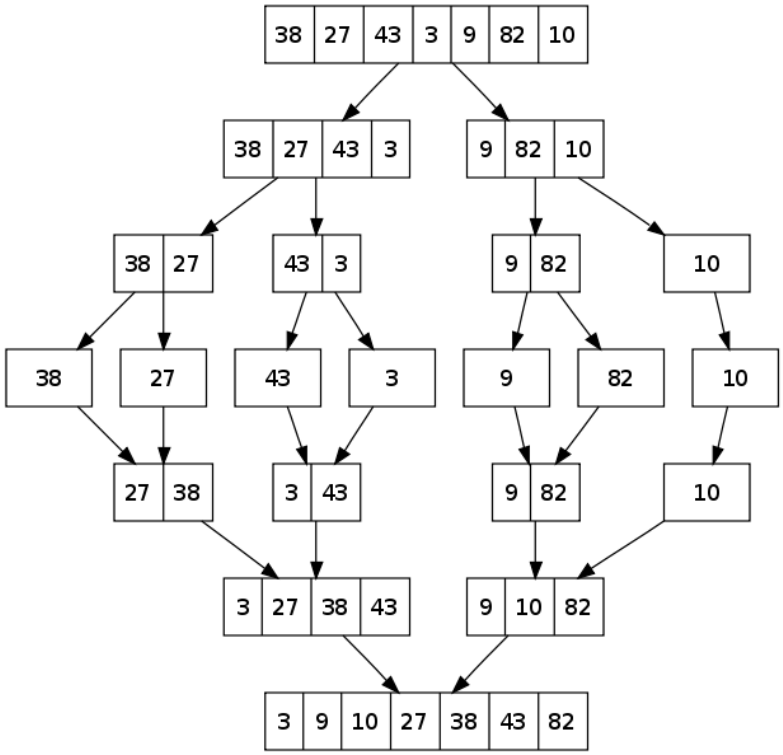

Figura 9. Ejemplo de ordenamiento de una secuencia mediante merge sort.

Como es posible ver, la función se llama recursivamente a sí misma dos veces y posteriormente ordena los resultados. El código de la implementación serial de este algoritmo es el siguiente:

```
int *mergesort(int *A, int n){
      if ( n == 1 ){
            return A;
      }
      else{
            int *C;
            int *B;
            C = mergesort(A+n/2, n-n/2);
            B = mergesort(A, n/2);
            int *D = merge(B, C, n/2);
            return D;
      }
}
```

Como es posible ver al analizar este código, la función *merge* es la que de hecho realiza el ordenamiento de cada uno de los elementos en la lista, mientras que la función *mergesort* se encarga de simplificar el problema. El código para la función *merge* se presenta a continuación:

```
int *merge(int *A, int *B, int n){
      int *C = new int[n*2];
      for ( int i = 0, j = 0; i < n || j < n;){
            if ( (A[i] < B[j] || j >= n) && i < n){
                  C[i+j] = A[i];
                  i++;
                  continue;
            }
            else{
                  C[i+j] = B[j];
                  j++;
                  continue;
            }
      }
      for ( int i = 0; i < n*2; i++)
            A[i] = C[i];
      delete C;
      return A;
}
```

- Complejidad de la implementación serial

Para calcular la complejidad del algoritmo, primero es necesario calcular la complejidad de la función *merge*, que es parte del algoritmo.

Como vemos, esta función realiza varias operaciones; primero traslada los elementos de los arreglos a un nuevo arreglo, y posteriormente copia los elementos del nuevo arreglo al arreglo inicial; luego el arreglo nuevo es eliminado. Esto es realizado en dos ciclos `for`, que van desde 0 hasta 2n. Es decir que la complejidad temporal de la función *merge* es $O(n)$.

Por otro lado, la complejidad temporal de la función *mergesort* es un tanto más sencilla. Tras realizar algunas operaciones, la función se llama a sí misma y luego la función *merge*; es decir que la recursión para la complejidad es:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

La derivación de la fórmula general para esta recursión es sencilla. La fórmula general de la complejidad temporal de la función mergesort es:

$$T(n) = O(n \log n)$$

Es decir que el algoritmo, al igual que *quick sort*, tiene el orden de complejidad temporal más bajo entre los algoritmos de ordenamiento. La complejidad temporal del algoritmo es $O(n \log \mathrm{n})$.

Paralelización del algoritmo

La paralelización del algoritmo de ordenamiento por mezcla es bastante inmediata; sabemos que la secuencia sobre la que operan las llamadas recursivas a las funciones *merge* y *mergesort* es un arreglo, y cada llamada de la función opera sobre cierta sección del arreglo. Es decir que las carreras de datos no son un problema de este programa, y no es necesario realizar reingeniería sobre ellas. El código de la implementación paralela de la función *mergesort* resulta ser casi igual a su versión serial. Se presenta a continuación:

```
int *mergesort(int *A, int n){
      if ( n == 1 ){
            return A;
      }
      else{
            int *C;
            int *B;
            B = cilk_spawn mergesort(A, n/2);
            C = cilk_spawn mergesort(A+n/2, n-n/2);
            cilk_sync;
            int *D = merge(B, C, n/2);
            return D;
      }
}
```

El código es prácticamente igual. Con la diferencia de las sentencias de creación de tareas y sincronización.

- Trabajo y profundidad

Dado que el código es casi idéntico, con la única diferencia de la creación de las tareas en paralelo, en vez de su ejecución en serie, el trabajo del algoritmo es igual a su complejidad temporal, es decir $W(n) = O(n \log n)$.

La profundidad puede ser calculada con una recursión muy sencilla:

$$D(n) = D\left(\frac{n}{2}\right) + O(n)$$

De aquí, la componente asintótica lineal domina la recursión; con esto se obtiene que:

$$D(n) = O(n)$$

De las formas generales para la profundidad y el trabajo, se calcula el paralelismo del algoritmo:

$$P(n) = O\left(\frac{n \log n}{n}\right) = O(\log n)$$

Este algoritmo tiene paralelismo de orden logarítmico. Es decir que el tamaño de la entrada debe ser mucho más grande que el número de procesadores disponibles para que cada núcleo del procesador sea efectivo, y la ecuación de tasa de desempeño se mantenga para este algoritmo.

De hecho, paralelismo de orden logarítmico es poco deseable. Es recomendable buscar un método para mejorar el orden del paralelismo del algoritmo; esto mejoraría especialmente su escalabilidad.

No tiene caso aumentar el trabajo del algoritmo para mejorar su paralelismo; eso sólo estaría provocando que el algoritmo fuera menos efectivo. Para mejorar el paralelismo del algoritmo es necesario **disminuir la profundidad**.

De la ecuación de derivación de la profundidad $D(n) = D\left(\frac{n}{2}\right) + O(n)$, podemos ver que el término $O(n)$ es el que domina la recursión. Para poder disminuir la profundidad del algoritmo es necesario disminuir el orden de complejidad de la función *merge*. Es decir que también es necesario paralelizar la función *merge*.

Reingeniería de la función *merge*

Ahora es necesario reinventar la función de mezcla de los arreglos, para que el algoritmo paralelo sea escalable, dado que un orden logarítmico de paralelismo no es suficiente. ¿Cómo se puede paralelizar el algoritmo de mezcla de dos arreglos ordenados?

En [3] se presenta una propuesta para paralelizar efectivamente el algoritmo de mezcla de dos arreglos ordenados, llamémosles A y B. Este nuevo algoritmo consiste en dividir uno de estos arreglos –para ser óptimo, el arreglo que sea más grande- en dos partes iguales; una vez dividido el primer arreglo, se divide al otro arreglo en dos partes, y se mezclan recursivamente. Este algoritmo utiliza al igual que los anteriores un método de divide y conquista para extraer paralelismo de la tarea.

Los pasos de la recursión de este algoritmo y el caso base se describen a continuación:

1. Dados dos arreglos ordenados que se desean mezclar, tomar el mayor, llámese A; y el menor, llámese B; donde A tiene tamaño N y B tiene tamaño M.
2. Si el tamaño de A o B es menor o igual a 1, realizar una mezcla serial.
3. Tomar el elemento A[N/2], y buscar en B el elemento i tal que B[i] < A[N/2] y B[i+1] >= A[N/2].

4. Mezclar recursivamente en paralelo los pares de arreglos A[0…N/2] con B[0…i], y A[N/2+1…N] con B[i+1…M].

Este algoritmo cuenta con todas las partes de un algoritmo recursivo. En el paso 3 se realiza la división del problema; luego en el paso 2 ocurre el caso base: si el tamaño de cualquiera de los arreglos es menor o igual a 1, entonces son mezclados de manera serial –ya que mezclar recursivamente en este caso degradaría el desempeño del algoritmo. Finalmente en el paso 4 se realizan las llamadas recursivas y cada una se encarga de unir la solución sobre el arreglo resultado.

El diagrama conceptual del algoritmo se muestra en la figura 10. Vale la pena observar que el algoritmo de búsqueda no se detalla aquí. El algoritmo de búsqueda en el arreglo será revisado después.

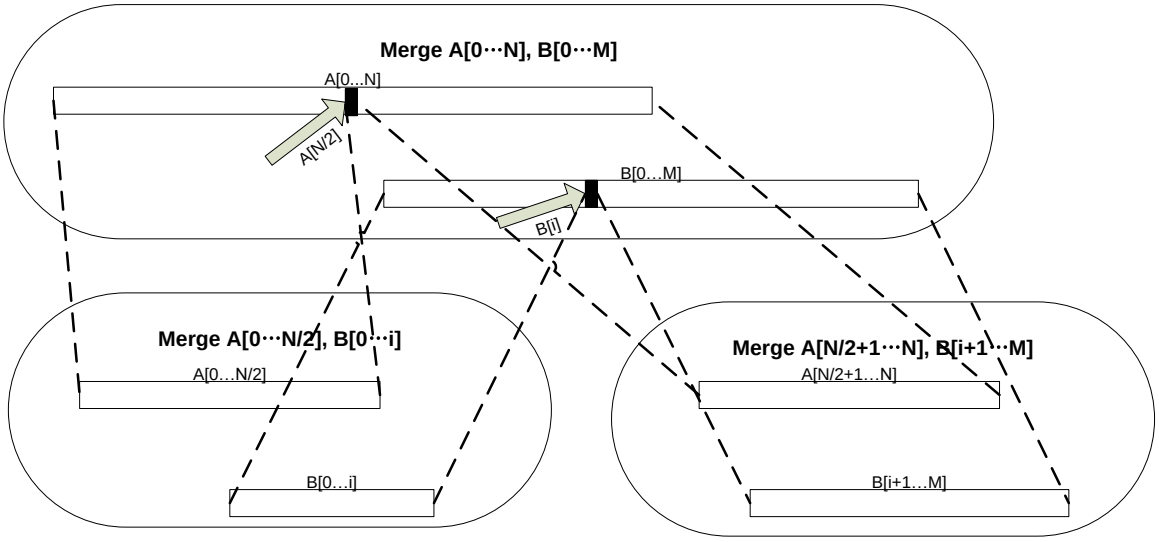

Figura 10. Primer nivel de recursión del algoritmo paralelo de mezcla.

El código para el algoritmo de mezcla de dos arreglos ordenados se presenta a continuación:

```
void merge(int *A, int *B, int szA, int szB, int *C){
      if ( szA <= 1 || szB <= 1 ){
            for( int i=0, j=0; i + j < szA+szB; ){
                  if ( i < szA && (A[i] < B[j] || j == szB)){
                        C[i+j] = A[i];
                        i++;
                  }
                  else if ( j < szB && (B[j] <= A[i] || i == szA)){
                        C[i+j] = B[j];
                        j++;
                  }
            }
      }
      else {
            int pivot;
            if ( szA <= szB ){
                  pivot = search(B[szB/2], A, szA);
                  cilk_spawn merge(A, B, pivot, szB/2, C);
      cilk_spawn merge(A+pivot, B+szB/2, szA-pivot, szB-szB/2, C+pivot+szB/2);
            }
            else{
                  pivot = search(A[szA/2], B, szB);
                  cilk_spawn merge(A, B, szA/2, pivot, C);
      cilk_spawn merge(A+szA/2, B+pivot, szA-szA/2, szB-pivot, C+pivot+szA/2);
            }
      }
}
```

Por otro lado, el algoritmo de búsqueda en el arreglo B es importante para mantener una buena profundidad. Necesitamos que este algoritmo sea rápido, ya que cualquier ganancia obtenida de paralelizar la mezcla puede ser mermada por una búsqueda ineficiente. Sabemos que ambos arreglos (A y B) se encuentran ordenados; el algoritmo de búsqueda puede aprovecharse de este hecho para optimizar la búsqueda.

El algoritmo más simple y de los más óptimos para búsqueda en arreglos ordenados es la búsqueda binaria. Se empieza buscando por el elemento en la mitad del arreglo, si no se encuentra el elemento, se toma la mitad hacia la que se encuentra el elemento (esto lo sabemos porque el arreglo está ordenado); luego se busca el elemento a la mitad de este subarreglo, y se continúa así hasta encontrarlo.

El código para el algoritmo de búsqueda binaria se presenta a continuación:

```
int search(int element, int *A, int szA){
      if( szA <= 1 )
            if ( A[0] < element )
                  return 1;
            else
                  return 0;
      if( A[szA/2-1] <= element && A[szA/2] > element )
            return szA/2;
      else if (A[szA/2] > element)
            return search(element, A, szA/2);
      else
            return (szA/2+search(element, A+szA/2, szA-szA/2));
}
```

- Trabajo y profundidad del nuevo algoritmo de mezcla

El análisis de este algoritmo de mezcla es un tanto complicado, ya que no es posible predecir la proporción en que el arreglo B será dividido. El caso óptimo es que el elemento B[i] se encuentra exactamente a la mitad del arreglo B, esto permite maximizar las recursiones y el paralelismo; sin embargo, éste no es el caso más común.

Para obtener el trabajo y la complejidad del algoritmo de mezcla, primero es necesario analizar el algoritmo de búsqueda. El código de búsqueda es muy sencillo. Como vemos, este código funciona recursivamente, buscando en el elemento medio del subarreglo que se le asigna buscar. Es un algoritmo serial, así que sólo nos interesa calcular su complejidad temporal.

Cada llamada al algoritmo realiza algunas comprobaciones, y luego una llamada recursiva sobre un subarreglo de la mitad del arreglo anterior. En el mejor caso, el programa encuentra al elemento en la primera recursión; en el peor caso, es necesario realizar cierta cantidad de llamadas recursivas antes de dar con el elemento. Esta cantidad de llamadas está en función del tamaño del arreglo, y es el orden de la complejidad del peor caso. La ecuación de recursión para esto es:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Es muy fácil derivar una forma general de esta ecuación. La forma general es la siguiente:

$$T(n) = O(\log n)$$

Es decir que la búsqueda es un algoritmo bastante óptimo, aún para el peor caso. En general, el caso promedio para la ejecución de este algoritmo es de complejidad $O(\log n)$.

Una vez que hemos analizado el algoritmo de búsqueda, queda analizar el algoritmo de mezcla. El código se divide básicamente en dos secciones. El caso base y las llamadas recursivas. A continuación se presenta el caso base:

```
      if ( szA <= 1 || szB <= 1 ){
            for( int i=0, j=0; i + j < szA+szB; ){
                  if ( i < szA && (A[i] < B[j] || j == szB)){
                        C[i+j] = A[i];
                        i++;
                  }
                  else if ( j < szB && (B[j] <= A[i] || i == szA)){
                        C[i+j] = B[j];
                        j++;
                  }
            }
      }
```

En el caso base, ya que uno de los dos arreglos es de tamaño 1, no vale la pena seguir trabajando recursivamente en él, ya que el paralelismo que se puede encontrar es muy bajo. Por esto se realiza una mezcla en serie de los dos arreglos.

Para las recursiones el análisis es más complejo. Primero vale la pena mencionar por qué queremos dividir el arreglo más grande a la mitad, y no el más pequeño: resulta que queremos hacer que el tamaño del problema disminuya tan rápido como sea posible, esto significa que queremos que ambos arreglos se hagan pequeños al menos en un factor de 2 cada recursión. Esto nos permite obtener un buen comportamiento logarítmico.

```
if ( szA <= szB ){
      pivot = search(B[szB/2], A, szA);
      cilk_spawn merge(A, B, pivot, szB/2, C);
      cilk_spawn merge(A+pivot, B+szB/2, szA-pivot, szB-szB/2, C+pivot+szB/2);
      cilk_sync;
}
```

Nuevamente se crean tareas que llaman funciones que se ejecutan en paralelo. Las líneas donde se realizan las llamadas recursivas como `cilk_spawn merge(A, B, pivot, szB/2, C);` se encargan de mezclar en orden cada una de las secciones de los arreglos que queremos mezclar. Por supuesto, es necesario sincronizar las tareas antes de concluir la mezcla, para prevenir resultados no determinísticos.

El análisis del algoritmo es complicado; para realizarlo, asumimos que en promedio, el arreglo B será dividido no a la mitad –porque ése sería el caso ideal-, sino en la cuarta parte, dividiendo así el arreglo en uno de M/4 elementos y otro de 3M/4 elementos. Dadas estas consideraciones, calculemos la profundidad:

Dado que buscamos la cadena más larga de dependencias, tenemos que tomar el peor caso; es decir, el caso donde el arreglo B sea dividido en ¾. Recordemos que además de la recursión, el algoritmo realiza una búsqueda binaria con complejidad temporal $O(\log n)$. Tomando todo esto en consideración, se obtiene la siguiente ecuación de recursión:

$$D(n) = D\left(\frac{3}{4}n\right) + O(\log n)$$

Aplicando el método maestro para la resolución de la recursión obtenemos una función logarítmica también del término recursivo. Esto resulta en una forma general de la siguiente forma:

$$D(n) = O(\log^2 n)$$

Este resultado es muy bueno. Mejora la profundidad respecto de la primera implementación. Esto es bueno para el algoritmo; ahora sólo queda verificar que el trabajo no haya aumentado sustancialmente debido a las adaptaciones que se hicieron al mezclar.

Para calcular el trabajo, sabemos que el problema se divide en dos partes de distintos tamaños. Dado que no sabemos la proporción, inicialmente le llamaremos $\propto$, y entonces la recursión para el trabajo del algoritmo es la siguiente:

$$W(n) = W(\propto n) + W((1-\propto) \cdot n) + O(\log n)$$

Esta recursión parece ser difícil de resolver. De hecho, el método maestro no es la herramienta adecuada para resolverla. Es necesario ocupar el método de sustitución.

1. Primero necesitamos suponer una solución general. La más sencilla, y la mejor es la complejidad del algoritmo serial de mezcla, es decir $W(n) = O(n)$. Para realizar el análisis es necesario utilizar un truco existente en el método de sustitución, que consiste en agregar un término de orden inferior para poder simplificar los términos existentes en la ecuación, y que no permiten que se derive una forma general correcta. Partiendo de esto, realizamos la siguiente suposición:

2. $W(n) = an - b \log n.$

3. Luego, sustituimos los valores de W(n) en el lado derecho:

$$W(n) = a(\propto n) + a((1-\propto) \cdot n) - b \log(\propto n) - b \log((1-\propto) \cdot n) + O(\log n)$$

4. De agrupar términos comunes, obtenemos que:

$$W(n) = an - b(\log(\propto n) + \log((1-\propto) \cdot n)) + O(\log n)$$

Por propiedades de los logaritmos:

$$W(n) = an - b(\log \propto + \log n + \log(1-\propto) + \log n) + O(\log n)$$

De ahí, los logaritmos de $\propto$ son constantes, y al utilizar notación asintótica se vuelven irrelevantes. Para los términos logarítmicos, escogemos una b suficientemente grande para eliminar el término $O(\log n)$; además, aislamos los términos que buscamos, y buscamos cancelar los que no:

$$W(n) = an - b \log n - (b(\log n + \log \propto + \log 1-\propto) - O(\log n))$$

5. Con esto, al utilizar notación asintótica, el término lineal domina el resto de la ecuación, por lo que el trabajo es de orden n:

$$W(n) = O(n)$$

Esto es excelente. Entonces, este algoritmo recursivo de mezcla de dos arreglos ordenados disminuye la profundidad del arreglo de ordenamiento, sin aumentar su trabajo, es decir que el nuevo algoritmo sigue siendo efectivo.

- Trabajo y profundidad del algoritmo de ordenamiento

Dado que el código es idéntico al primer ejemplo de ordenamiento paralelizado, con la única diferencia de la mezcla en paralelo, el trabajo del nuevo algoritmo de ordenamiento es igual a la complejidad temporal de la versión serial, es decir $W(n) = O(n \log n)$.

La profundidad puede ser calculada con una recursión muy sencilla, que ahora debe incluir la profundidad del nuevo algoritmo de mezcla, es decir:

$$D(n) = D\left(\frac{n}{2}\right) + O(\log^2 n)$$

Luego entonces:

$$D(n) = O(\log^3 n)$$

El paralelismo del algoritmo cambia, y es ahora de la siguiente forma:

$$P(n) = O\left(\frac{n \log n}{\log^3 n}\right)$$

Es decir que:

$$P(n) = O\left(\frac{n}{\log^2 n}\right)$$

Es posible observar que el paralelismo ha aumentado a un orden mucho mejor para escalar el algoritmo hacia más procesadores. Ya se verá en las pruebas los resultados que se obtengan al ejecutar ambos algoritmos en distintos números de núcleos.

4. El problema del vendedor viajero

Versión serial del algoritmo

El problema del vendedor viajero es un problema clásico de la computación; que además también se estudia en investigación de operaciones y optimización, existen muchos problemas prácticos que pueden ser modelados como problemas de agente viajero; por ejemplo, problemas de logística y manufactura. Es uno de los problemas más estudiados en matemática computacional.

Dada una colección de ciudades y el costo de viajar entre cada par de ellas, el problema del vendedor viajero es encontrar la ruta más corta –o barata- para recorrer todas las ciudades una vez y volver al punto inicial. En este trabajo, los costos de viajar entre cada par de ciudades son simétricos; es decir que el costo de ir desde X hasta Y es igual al costo de ir desde Y hasta X.

Existen varias formas de obtener la solución a este problema; e incluso varias formas de modelarlo. El método de fuerza bruta para solucionar este problema consiste en probar todas las permutaciones válidas del conjunto de ciudades y seleccionar aquella que tenga menor distancia recorrida.

La resolución del problema por fuerza bruta debe probar $n!$ permutaciones, donde n es el número de ciudades. Es decir que la complejidad de tiempo de ejecución del algoritmo serial para resolver el problema del vendedor viajero por fuerza bruta es $O(n!)$. La resolución del problema por fuerza bruta tiene un orden de complejidad demasiado alto, así que el método se vuelve impráctico para una cantidad moderada de ciudades.

Otros métodos de la inteligencia artificial que permiten solucionar el problema cuando el espacio de soluciones posibles se vuelve demasiado grande para los métodos de fuerza bruta, son los métodos de búsqueda local; también conocidos como métodos de mejora iterativa.

Los métodos de mejora iterativa inician en una configuración aleatoria de la solución del problema –en este caso, un orden aleatorio para recorrer las ciudades-; partiendo de esa solución, se genera una 'vecindad', es decir, un conjunto de soluciones que resultan de modificar ligeramente la solución inicial– por ejemplo, intercambiar un par de ciudades en el orden de recorrido-; una vez generada la vecindad, se selecciona un miembro de esta vecindad, basado en alguna función de evaluación, y se itera en el algoritmo, ahora con ese miembro de la vecindad como solución actual.

Existen varios algoritmos de mejora iterativa: *Hill Climbing, Randomized Hill Climbing, Simulated Annealing*, Algoritmos genéticos, etc. [4] Entre estos métodos existe uno menos conocido, que se utiliza en este trabajo; es un algoritmo de búsqueda menos conocido, pero que ha dado muy buenos resultados en aplicaciones prácticas: *Tabu Search*. [5]

La búsqueda tabú es un algoritmo de mejora iterativa en el que, para poder mejorar la exploración del espacio de soluciones, se mantiene no sólo la información de la solución actual; sino que también se mantiene una lista de soluciones exploradas recientemente para evitar que el algoritmo quede estancado iterando alrededor de máximos locales; es decir que a diferencia de otros algoritmos de búsqueda local, el algoritmo de búsqueda tabú utiliza una memoria para recordar algunos estados anteriores del proceso de solución.

Un algoritmo de búsqueda tabú tiene tres estrategias principales:

- Estrategia de prohibición. Esta estrategia es la que controla qué es lo que entra y cómo es que entra a la lista de movimientos tabú.
- Estrategia de liberación. Esta estrategia es la que controla qué es lo que sale y porqué es que sale de la lista de movimientos tabú.
- Estrategia a corto plazo. Esta estrategia es la que determina los movimientos que se hacen en el espacio de soluciones; ésta es la que se encarga de integrar todo el algoritmo para buscar soluciones factibles para el problema.

Planteamiento de las estrategias para el algoritmo del vendedor viajero

El algoritmo del vendedor viajero y su resolución mediante búsqueda tabú se realizan partiendo de soluciones generadas aleatoriamente –todas con la misma ciudad inicial, e iterando a partir de ahí para buscar soluciones con menor distancia total viajada. Es decir que el proceso en la estrategia a corto plazo es más o menos así:

1. Generar una solución aleatoria
2. Generar una vecindad de soluciones con cambios menores en la solución actual
    a. La vecindad consiste de todas las soluciones que pueden ser generadas al intercambiar dos ciudades en el orden de visita de la solución actual
3. De esa vecindad, seleccionar la solución con la distancia recorrida más corta y que no se encuentre en la lista de soluciones tabú
4. Agregar la solución actual a la lista de soluciones tabú –si es necesario sacar otra solución para agregar la actual, hacerlo.
5. Si la solución actual es mejor que la mejor solución conocida hasta ahora, actualizar la mejor solución conocida hasta ahora.
6. Si se ha iterado suficiente, o si ha habido suficientes iteraciones sin ninguna mejora en la solución, devolver la mejor solución conocida hasta ahora.
7. Si no se ha iterado suficiente, ir al paso 2.

Esto permite reducir la vecindad a un tamaño de orden $O(n)$, donde n es la cantidad de ciudades en nuestro 'mundo'; al reducir la vecindad, podemos reducir el tiempo que se pasa generando y evaluando las soluciones disponibles en ésta. Dado que para evaluar una solución es necesario calcular la distancia que recorre, es necesario calcular la distancia entre cada una de las ciudades en el orden en que son recorridas, es decir un cálculo de tipo $O(n)$; esto significa que la generación de la vecindad y obtención del vecino con la menor distancia es de orden $O(n^2)$.

Ahora, para las estrategias de prohibición y liberación, es importante disminuir el costo de búsqueda de las soluciones en una lista común y corriente. Dado que queremos una búsqueda rápida, la estructura que nos permite realizar una búsqueda rápida y efectiva es un hash table; con una función de hash para cada solución que permita diferenciarla de soluciones que puedan ser vecinas.

Cuando una solución sea asignada a una posición de la lista de soluciones tabú –que es un hash table- que ya se encuentre ocupada por otra solución es que se realiza la liberación de soluciones de la lista tabú. La nueva solución es agregada a la lista tabú y la solución que ocupaba la posición es eliminada de la lista tabú.

La codificación del programa

En la codificación del programa existen varias 'entidades', que son representadas en forma de clases. Las clases que se utilizan en el programa son las siguientes:

- La clase mundo (`World`). En nuestro mundo hay una serie de ciudades que queremos visitar. El mundo tiene un tamaño. Este mundo tiene una lista de soluciones tabú que no se permiten. Es la encargada de solucionar el problema.
- La clase ciudad (`City`). Una ciudad tiene una posición dentro del mundo. También puede calcular la distancia entre sí misma y otra ciudad.
- La clase solución (`Solution`). Una solución es específica para un mundo. Tiene una lista de las ciudades en el orden en que deben ser recorridas. Es capaz de generar una ruta aleatoria en este mundo. Tiene una distancia total, que es la que se necesita andar para recorrerla completamente. Puede generar su propia vecindad y obtener su 'mejor vecino'. Es capaz de compararse con otra solución y determinar si son iguales.
- La clase de lista tabú (`TabuHash`). Mantiene una lista de soluciones tabú. Puede obtener el valor de hash de una solución. Puede determinar si una solución es una solución tabú. Puede insertar una nueva solución a la lista tabú, y puede eliminar todas las soluciones de la lista tabú. Puede calcular la posición que una solución debe tener en la lista de soluciones tabú; es decir, el valor hash.

Dada la descripción anterior de las clases, sus declaraciones iniciales se presentan a continuación. Vale la pena observar cómo en general sus características se traducen en atributos y sus capacidades en métodos:

La clase mundo. **World.h**

```
class World{
public:
      World(int nCities, int new_length, int new_width);
      Solution* Solve(int cycles, int aspiration, int repetitions, int seed);
      int numberCities;
      City **cities;
      int length;
      int width;
      bool PopulateRandomCities();
      TabuHash *hasher;
};
```

Esta clase tiene los siguientes atributos y métodos:

- `numberCities`. Es el número de ciudades en el mundo. Un entero.
- `**cities`. Es un apuntador a un arreglo de ciudades. Es la lista de ciudades en el mundo.
- `width, length`. Son las dimensiones del mundo. Se necesitan para posicionar las ciudades.
- `*hasher`. Es un apuntador a la lista de soluciones tabú.
- `Solve(int, int, int, int)`. Es el método que itera para encontrar soluciones mínimas en el mundo. Devuelve la mejor solución que encuentre. Los argumentos serán revisados al revisar el código.
- `PupulateRandomCities()`. Es la función encargada de generar una lista de ciudades aleatoria.

La clase ciudad. **City.h**

```
class City{
public:
	City(float new_x, float new_y);
	float x;
	float y;

	float distanceTo(City *goingTo);
};
```

Sus atributos y métodos son:

- `x, y.` Son las coordenadas en que se encuentra la ciudad. Son calculadas al generarla, y son menores al tamaño del mundo.
- `distanceTo(City*).` Devuelve la distancia entre la ciudad actual y la que se pasa como argumento.

La clase solución. **Solution.h**

```
class Solution{
public:
	Solution(World *theWorld);
	int numberCities;
	World *myWorld;
	City **path;
	float pathLength;
	bool equals(Solution *compareTo);
	bool updateLength();
	bool RandomPath(int seed);
	bool swapCities(int city1, int city2);
	Solution *getBestNeighbor();
	Solution *cloneSolution();
private:
	void initializeSolution();
};
```

Los atributos y métodos de esta clase son:

- `numberCities`. Al igual que en la clase mundo, es un entero con la cantidad de ciudades.
- `**path.` Es una lista de ciudades en el orden en que son recorridas en esta solución.
- `pathLength.` Es la distancia recorrida al recorrer el mundo mediante esta solución.
- `*myWorld` . Es el mundo al que está asociado esta solución.
- `equals(Solution*).` Devuelve si la solución actual es igual a la solución que se le pasa como argumento.
- `updateLength().` Calcula la distancia recorrida en esta solución y actualiza el valor de `pathLength.`
- `RandomPath(int).` Genera una ruta aleatoria. El argumento se utiliza como semilla para el generador de números aleatorios.

- `swapCities(int, int)`. Intercambia el orden en que se visitan las ciudades pasadas como argumento y actualiza el valor de `pathLength`.
- `getBestNeighbor()`. Devuelve el miembro de la vecindad con la menor distancia que no se encuentre en la lista tabú.
- `clonseSolution()`. Devuelve una solución igual a la actual.

La clase lista tabú. **TabuHash.h**

```
class TabuHash{
public:
	TabuHash(int n);
	const int size;
	Solution **tabuSolutions;
	int HashSolution(Solution *hashMe);
	bool admits(Solution *testMe);
	bool insertTabuSolution(Solution *tabuSolution);
	bool clean();
};
```

Sus atributos y métodos son:

- `size`. Es la cantidad de elementos que se guardan en la lista de soluciones tabú. Normalmente son 256 para poder ser direccionada con un byte.
- `**tabuSolutions`. Es un puntero al arreglo de soluciones que son marcadas como tabú.
- `HashSolution(Solution*)`. Devuelve el valor hash de la solución que se le pasa como argumento.
- `admits(Solution *)`. Devuelve `true` si la solución que se le pasa como argumento no se encuentra en la lista de soluciones tabú. Devuelve `false` en cualquier otro caso.
- `insertTabuSolution(Solution*)`. Agrega la solución que se le pasa como argumento a la lista de soluciones tabú.
- `clean()`. Vacía la lista de soluciones tabú. Cualquier solución es aceptada.

Finalmente, el código núcleo del programa se muestra a continuación. El código completo está disponible en el disco apéndice.

```
void main(){
	Solution *bestFoundSolution;
	World* myWorld = new World(100, 300, 300);
	//Un mundo de 100 ciudades y 300x300 de tamaño.
	myWorld->PopulateRandomCities();

	bestFoundSolution = myWorld->Solve(1000, 650, 10, 0);
	//1000 iteraciones máximas
	//650 iteraciones sin mejora en la solución
	//Partiendo de 10 soluciones generadas aleatoriamente
	//0 es la semilla para la generación de números aleatorios
}
```

Luego, la función de resolución es la siguiente:

```
Solution *World::Solve(int cycles, int aspirations, int repetitions, int seed){
	int sadness;
	Solution *mySolution;
	Solution *bestSolution, *veryBestSolution;
	for(int j = 0; j < repetitions; j ++){
	//repetitions es la cantidad de veces que creamos una nueva solución
	//aleatoria e iteramos para mejorarla
		sadness = 0;
		mySolution = new Solution(this);
		mySolution->RandomPath(seed + j);
		mySolution->updateLength();
		bestSolution = mySolution->cloneSolution();
		hasher->insertTabuSolution(mySolution);
		if ( j==0 ) veryBestSolution = mySolution->cloneSolution();
		for(int i=0; i<cycles; i++){
			Solution *neighbor = mySolution->getBestNeighbor();
			if( hasher->admits(neighbor) ){
				mySolution = neighbor;
				hasher->insertTabuSolution(mySolution);
				//Insertamos la solución actual en la lista tabú
				if( mySolution->pathLength < bestSolution->pathLength ){
		//Se mejora la mejor solución de esta iteración
					delete bestSolution;
					bestSolution = neighbor->cloneSolution();
					sadness = 0;
					//Reinician las aspiraciones
				}
				else{
			//No se mejoró la solución en esta iteración. Eso entristece.
					sadness++;
					if ( sadness > aspirations ) break;
			//Si la cantidad de iteraciones en que no se mejora la soluc.
			//es muy alta, nos resignamos y pasamos a la siguiente
			//repeticion
				}
			}
		}
		if( bestSolution->pathLength < veryBestSolution->pathLength ){
		//Si la solución actual es mejor que la mejor, actualizamos.
					delete veryBestSolution;
					veryBestSolution = bestSolution->cloneSolution();
				}
		hasher->clean();
		//Se limpia la lista para iniciar una nueva solución
	}
	return veryBestSolution;
}
```

La complejidad del algoritmo

Los algoritmos de búsqueda local no tienen una complejidad específica, ya que en general son aleatorios y no garantizan la solución óptima. Tanto puede ser que la solución óptima nunca sea obtenida, como puede ser que sea generada aleatoriamente al inicio de una iteración. En general, el iterar varias veces sobre una solución permite generar nuevas soluciones que mejoran mucho la solución inicial.

Al realizar más iteraciones con nuevas soluciones aleatorias se consigue aumentar la probabilidad de encontrar la solución óptima; es aquí donde el aumento de la capacidad de un algoritmo a manera de algoritmo paralelo puede beneficiarnos; ya que la obtención de una solución optimizada a partir de cada solución aleatoria es completamente independiente a cualquier otra iteración.

Vale la pena señalar que el algoritmo incurre algunos costos que sí son medibles; por ejemplo la obtención del mejor vecino dada una solución tiene complejidad $O(n^2)$, y el cálculo del valor hash de una solución es de complejidad $O(n)$. Estos costos son bastante altos para las funciones que realizan; y hacen que el algoritmo tenga ciertos costos implícitos que aumentan con la cantidad de ciudades que se agreguen al problema, aunque su costo total no pueda ser calculado.

Recordemos que la solución por fuerza bruta del problema tiene complejidad $O(n!)$, así que deja de ser factible ejecutarla incluso con muy pocas ciudades. Los algoritmos de búsqueda local permiten encontrar buenas soluciones para problemas que son demasiado complejos para ser resueltos por fuerza bruta.

Paralelización del algoritmo

La paralelización de este algoritmo es distinta a la de los algoritmos anteriores. En este caso, no es necesario buscar paralelismo dentro de las rutinas del algoritmo, sino aprovechar los recursos paralelos disponibles para aumentar la probabilidad de encontrar la solución óptima al problema sin aumentar el tiempo de ejecución del programa.

Para aumentar la probabilidad, necesitamos aumentar la cantidad de soluciones aleatorias que se generan y se optimizan, porque alguna de éstas podría encontrarse en el área de la solución óptima. La disponibilidad de más recursos paralelos permite crear nuevas tareas que optimicen más soluciones en busca de la solución óptima.

Esto significa que se crearán más soluciones en paralelo y se tomará de entre éstas la que tenga la menor distancia recorrida. Es necesario reescribir algunos aspectos del programa serial que fueron realizados considerando que se calculen soluciones una por una.

Algunas de las secciones que es necesario reescribir son las siguientes:

- Se necesita la existencia de varias listas de soluciones tabú por cada tarea paralela que esté optimizando soluciones.
- Es necesario reducir las nuevas soluciones obtenidas en paralelo y devolver sólo aquella que tenga la distancia mínima.

Es interesante observar un detalle de estas características: El tener N hilos de ejecución, existirán N listas tabú, y puede ser que existan más 250 soluciones por cada una de estas listas, y cada solución es una lista de varios punteros; esto significa que conforme la cantidad de hilos de ejecución con que el algoritmo es ejecutado aumenta, la memoria que éste ocupa aumenta linealmente; esto puede resultar problemático al escalar hacia muchos hilos de ejecución.

La nueva declaración de la clase `World`, con varias tablas tabú es la siguiente:

```
class Solution;
class City;
class TabuHash;

class World{
public:
	World(int nCities, int new_length, int new_width, int
parallelSolvers);
	Solution* Solve(int cycles, int aspiration, int repetitions, int seed,
int instance);
	int numberCities;
	City **cities;
	int length;
	int width;
	bool PopulateRandomCities();
	TabuHash **hasher;
};
```

También es necesario incluir los argumentos `parallelSolvers` en el constructor y `instance` en el método `Solve` para indicar la cantidad de listas tabú que es necesario tener, y la lista que el método va a utilizar para hacer la búsqueda, respectivamente.

Dado que se obtienen varias soluciones y es necesario seleccionar la mejor, el método principal también tiene que ser reescrito para crear las tareas paralelas y para la reducción de la solución con distancia mínima. Es necesario agregar el siguiente segmento de código:

```
for(int i=0; i<workers; i++){
	solutions[i] = cilk_spawn myWorld->Solve(1000, 650, repetitions,
	i*repetitions, i);
}
cilk_sync;
for(int i=0; i<workers; i++){
	if(i==0){
		chosenSolution = solutions[i];
		continue;
	}
	if(chosenSolution->pathLength > solutions[i]->pathLength)
		chosenSolution = solutions[i];
}
```

La creación de las tareas paralelas y el ciclo de reducción son el trabajo extra en el algoritmo, que puede provocar que la ejecución sea más lenta. Sin embargo, son pasos relativamente insignificantes a comparación con la ejecución del resto del algoritmo.

También es necesario realizar algunas adaptaciones menores al código de los métodos del programa para mantener los datos y las listas tabú separados. El código completo se encuentra en el disco incluido.

Referencias

[1]. Drozdek, A. (2005). Data Structures and Algorithms in C++. (2a. ed.)
EE.UU.: Thomson Learning.

[2]. Intel Corporation.
(2009). Cilk++ Programmer's Guide.

[3] Kruskal, C.P.
(Octubre 1983). Searching, merging, and sorting in parallel computation.
IEEE Transactions on Computers, C–32(10):942–946.

[4]. Rosen, B. University of California Los Angeles.
(Septiembre 2009). Iterative improvement algorithms.
Revisado el 4 de Noviembre de 2010, de
http://www.cs.ucla.edu/~rosen/161/iterative_improvement_algorithms.pdf

[5]. Hertz, A., Taillard, E. and Werra, D.
(1992). A Tutorial on Tabu Search.
Revisado el 4 de Noviembre de 2010, de
http://www.cs.colostate.edu/~whitley/CS640/hertz92tutorial.pdf

Capítulo 4. Análisis comparativos del desempeño

En este capítulo se revisan los resultados de realizar análisis de tiempo de ejecución y de escalabilidad paralela de los algoritmos. Para esto se utilizan dos herramientas: Para el análisis de escalabilidad se utiliza el analizador de escalabilidad de Cilk++, y para el análisis de tiempo de ejecución se utiliza el Manycore Testing Lab.

El analizador de escalabilidad de Cilk++ es una herramienta que se incluye con Cilk++. Permite obtener una medida práctica y aproximada del paralelismo de un programa. Para obtener esta medida, el analizador observa la ejecución del programa en cuestión, y calcula el trabajo y la profundidad en términos de instrucciones en ensamblador. El trabajo se calcula como la cantidad total de instrucciones que son ejecutadas por el entorno de ejecución de Cilk++ al ejecutar el programa (es decir, instrucciones de planificación, reducción y las propias del programa). La profundidad es calculada como la cantidad promedio de instrucciones que deben ser ejecutadas en serie al ejecutar el programa. El cálculo de la cantidad promedio es complicado. El analizador hace estimaciones de límites superiores e inferiores, considerando que la planificación de las tareas se vuelva compleja, o muy sencilla.

Una vez que Cilk++ calcula la profundidad y el trabajo que realiza el programa, el paralelismo es el cociente de estas dos cifras.

El Manycore Testing Lab (MTL), como se describió en el capítulo 2, es una plataforma de alto desempeño provista por Intel para realizar investigación y desarrollar material para la enseñanza del cómputo paralelo. Es un servidor que cuenta con 32 núcleos de ejecución distribuidos en 4 chips. Cada núcleo cuenta con HyperThreading, así que el MTL puede ejecutar hasta 64 hilos al mismo tiempo. Las pruebas de tiempo de ejecución se realizan en el MTL, revisando cómo escala la velocidad del programa conforme se aumenta la cantidad de núcleos en que es ejecutado.

1. Cálculo recursivo de los números de Fibonacci

Análisis de escalabilidad

Entrada	Paralelismo
15	1.10
20	2.14
25	13.60
30	140.54
31	226.66
32	366.12
33	591.75
34	956.83
35	1,547.44
36	2,501.79

Según el análisis teórico, el paralelismo del algoritmo paralelo para calcular los números de Fibonacci es $P(n) = O\left(\frac{2^n}{n}\right)$; es decir que en general, el paralelismo aumenta exponencialmente con el tamaño de la entrada. A continuación se presentan los resultados de paralelismo obtenidos por el analizador de escalabilidad de Cilk++ para distintas entradas del programa:

Como es posible ver de la tabla, conforme la entrada se vuelve más grande, el paralelismo del algoritmo se dispara. Parece tener forma exponencial, que es lo que buscamos.

Un detalle a tener en cuenta es que para una entrada menor a 20, el

paralelismo del programa cae por debajo de 2, y no vale más la pena paralelizarlo; ya que en general su ejecución no se acelera al incluir nuevas unidades de procesamiento. Afortunadamente, éste es un algoritmo fácil de paralelizar, sin embargo, en muchos casos puede ocurrir que no valga la pena paralelizarlo, pues no será posible mejorar el desempeño en un grado que amortice el costo de reprogramar.

A continuación, en la figura 11 se muestran los puntos de paralelismo según la entrada al programa. Es posible apreciar que conforme la entrada aumenta, el paralelismo se dispara. El algoritmo es muy escalable, y esto es lo que buscamos.

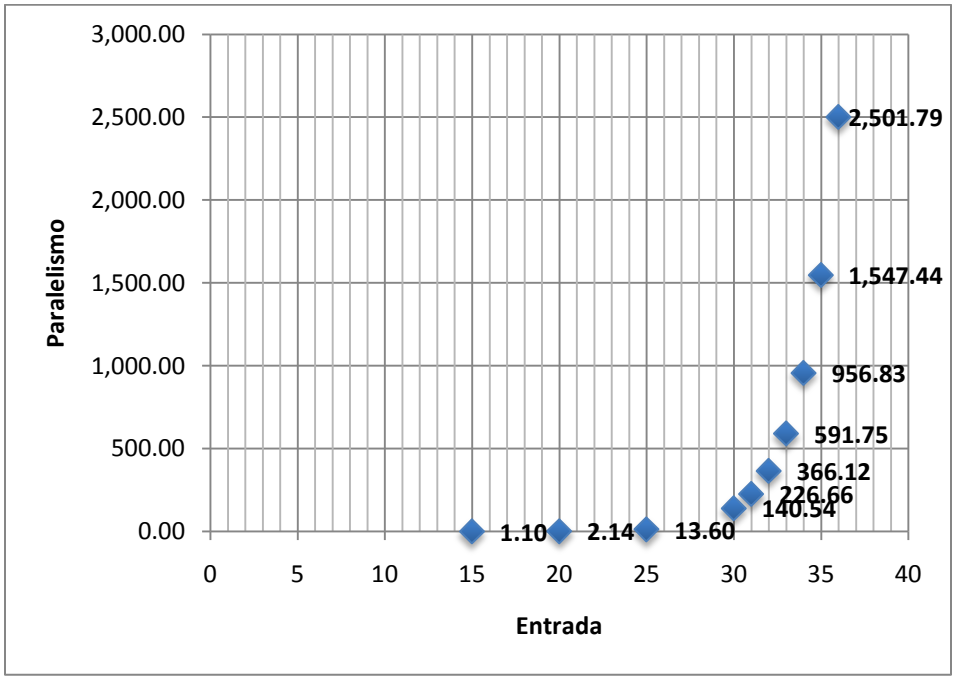

Figura 11. Gráfica del paralelismo del programa para calcular los números de Fibonacci.

Estos puntos muestran un comportamiento exponencial indiscutible, y se verifica la hipótesis: El paralelismo crece exponencialmente según la entrada.

Análisis de tiempo de ejecución

Dado que se busca comprobar la efectividad del modelado de trabajo-profundidad, para este algoritmo se tomó una entrada de 45, para disponer de la mayor cantidad posible de paralelismo. Y se midieron los tiempos de ejecución al ejecutar el programa con cierta cantidad de hilos de ejecución de sistema operativo, desde 1 hasta 64 –la cantidad de hilos que pueden ser ejecutados en el MTL al mismo tiempo.

En el apéndice 1 se presenta la tabla con los resultados del tiempo de ejecución del algoritmo de los números de Fibonacci.

La efectividad de este algoritmo, es decir, la tasa del tiempo de ejecución del programa paralelo en un solo hilo de ejecución entre el tiempo de ejecución de la versión serial del programa es de **0.6888**. Esta

cifra permite observar cómo afecta al desempeño del programa la creación de tareas de ejecución paralela. Esto habla del trabajo extra que debe realizar el programa para habilitar su ejecución en paralelo.

En la figura 12 se muestra la gráfica de aceleración de la ejecución del programa con respecto a la cantidad de hilos que fueron creados para ejecutarlo:

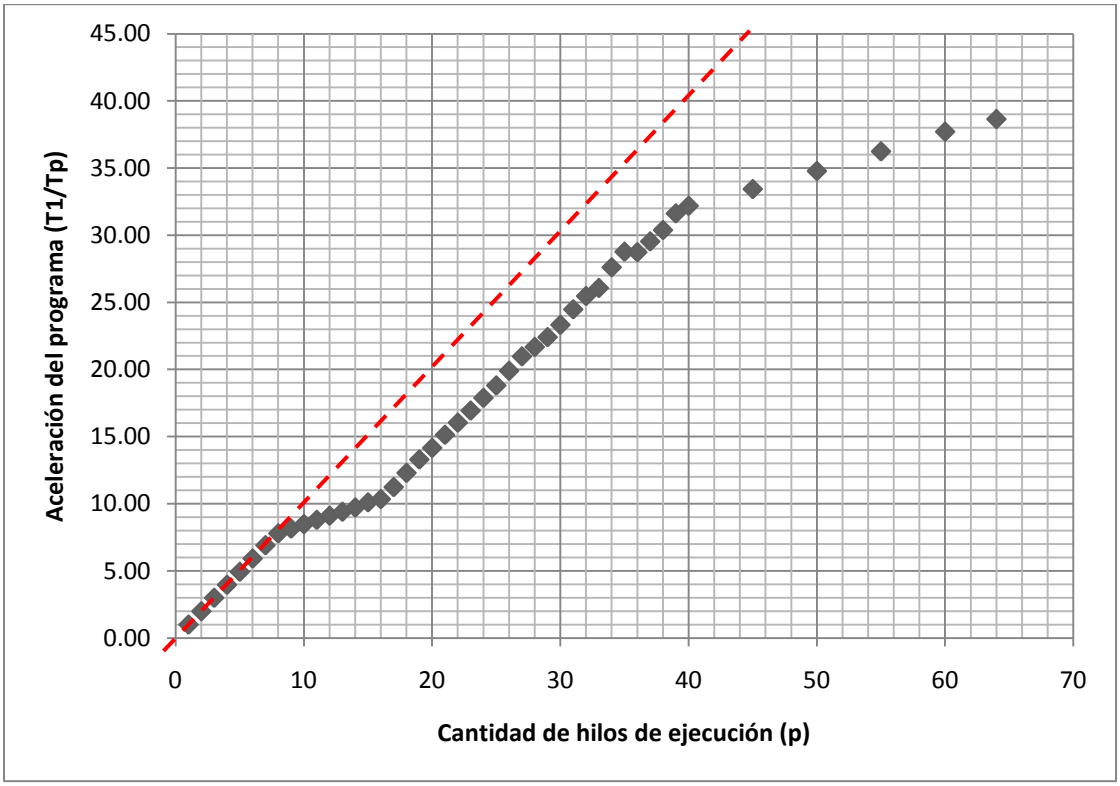

Figura 12. Gráfica de aceleración del programa contra cantidad de hilos de ejecución.

La línea punteada representa la ecuación de tasa de desempeño lineal; los puntos que se encuentran sobre esta línea son puntos en los que el modelo de trabajo profundidad representa precisamente el comportamiento del algoritmo.

De la figura, se puede observar que hay algunos puntos en ésta que se ajustan muy precisamente a la recta punteada; sin embargo, la mayor parte de los puntos tienen comportamientos de aceleración sublineal.

Para el análisis de los resultados, conviene dividir la gráfica en 4 partes, según el comportamiento de la gráfica de tiempos de ejecución:

1. La sección de ecuación de tasa de desempeño lineal. Las muestras obtenidas para 1, 2, 3, 4, 5, 6, 7 y 8 hilos de ejecución cumplen con la ecuación de tasa de desempeño lineal. En la tabla del apéndice 1 se puede apreciar que todas estas muestras tienen una eficiencia prácticamente de 1; es decir que todos los núcleos en que se ejecuta el programa son utilizados adecuadamente.

2. La sección de las muestras obtenidas a partir de 9 y hasta 16 hilos de ejecución tiene una pendiente inferior a las pendientes de sus secciones contiguas. Es probable que estos hilos hayan sido planificados dentro del mismo chip, compartiendo núcleos con los primeros 8; esto utilizando la capacidad de HyperThreading que tiene el procesador Xeon x7560, y disminuyendo la velocidad a la que cada hilo se ejecuta, aunque globalmente la aplicación aumentó de velocidad por los recursos compartidos. A partir de este periodo, la ecuación de tasa de desempeño lineal no se cumple para ningún caso en el resto de las muestras.
3. La sección media. Las muestras obtenidas a partir de 17 y hasta 40 hilos de ejecución tienen aceleración sublineal, pero pueden ser más o menos ajustadas a una recta con ordenada al origen negativa. Las eficiencias de esta sección están distribuidas con media **0.7612** y desviación estándar **0.056401**; para esta sección se mantuvo una mejora sublineal en el desempeño, pero en general con buena eficiencia.
4. La sección final. Las muestras obtenidas para más de 40 hilos de ejecución tienen muy mala tasa de aceleración por núcleo, y también muy malas eficiencias por debajo de **0.7**. Esto seguramente es debido al costo que es planificar una cantidad tan grande de hilos, así como la intercomunicación de todos los núcleos y la compartición de recursos al planificar la mayor parte de los hilos utilizando HyperThreading.

2. El problema de las N Reinas

Análisis de escalabilidad

Especialmente para los algoritmos para resolver el problema de las N reinas, la escalabilidad es un tema importante, ya que particularmente en la primera implementación hubo que aumentar el trabajo del algoritmo para lograr paralelismo.

El orden del paralelismo para ambas implementaciones es $O(n^{n-2})$, esto es mucho paralelismo. Lo que podría hacer la diferencia entre el paralelismo de estos dos algoritmos es la intercomunicación y paso de argumentos entre tareas. A continuación se muestran los resultados del análisis:

	Paralelismo	
Entrada	**Primera versión**	**Versión final**
7	1.93	1.11
8	5.07	1.41
9	16.99	2.89
10	75.55	9.00
11	370.98	41.37
12	2,208.15	202.93
13	12,810.65	1,178.94
14	83,084.18	6,731.10
15	N.D.	44,857.92

En la primera versión del programa, el paralelismo para una entrada de 15 no está disponible dado que no es factible hacer el análisis; ya que tomaría cerca de dos días ejecutar la aplicación y obtener un resultado de este análisis.

Sin embargo, estos resultados permiten verificar ambas hipótesis del paralelismo de estos programas:

i. El paralelismo es de orden exponencial; crece enormemente según la entrada. Es decir que el programa

es muy escalable.

ii. En la versión final del algoritmo se sacrifica paralelismo para ganar efectividad; aunque en el análisis teórico, ambos algoritmos tienen un paralelismo de orden $O(n^{n-2})$, el paralelismo práctico de la versión final es mermado. El motivo de esta merma es, seguramente, el costo que se incurre al tener que pasar una mayor cantidad de información a la tarea en forma de argumentos.

Como sabemos, en la primera versión, cada tarea `void putQueen(NodePosition*, int);` toma un puntero de 4 bytes y un entero de 4 bytes; es decir 8 bytes. Por otro lado, en la versión final, cada tarea `void putQueen(int, long, long, long);` toma cuatro enteros de 4 bytes; es decir 16 bytes en total. Esta diferencia podría parecer nimia y poco relevante, pero el paso de argumentos a una tarea es bastante más costoso que el simple paso de argumentos a una función, como expresan Blumofe, Leiserson *et al.* en [1]:

"Looking more carefully at the cost of a `spawn` in Cilk, we find that it takes a fixed overhead of 50 cyclesto allocate and initiallize a closure, plus about 8 cycles for each word argument. In comparison, a C function call […] takes 2 cycles of fixed overhead plus 1 cycle for each word argument. Thus, a `spawn` in Cilk is roughly an order of magnitude more expensive than a C function call."

Es decir que los argumentos extra no deben ser tomados a la ligera, y seguramente en este caso son los que causaron una diferencia tan marcada en el paralelismo real que se obtiene del programa. El costo extra de cada argumento pasado merma el desempeño del programa.

A continuación se presenta la gráfica del paralelismo de la primera versión y luego de la versión final del programa. Por motivos de escala, son presentadas en dos gráficas distintas; sin embargo vale la pena señalar que geométricamente, ambas gráficas tienen una forma muy similar; que surge dado que ambos algoritmos tienen el mismo orden de paralelismo, pero hay factores constantes que no son expresados por el análisis asintótico, y que actúan en este caso.

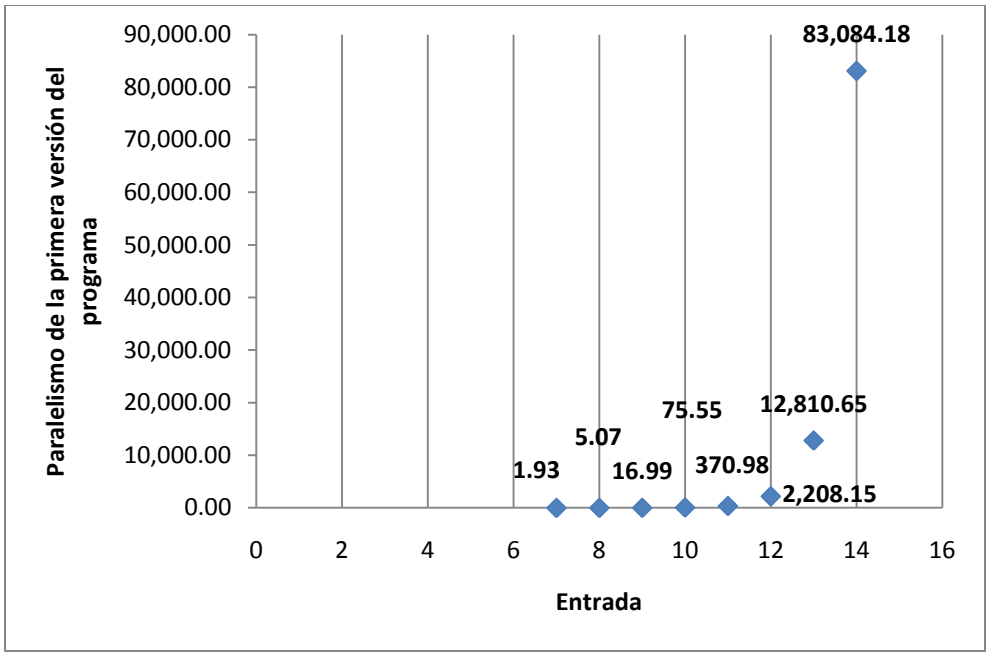

Figura 13. Gráfica de paralelismo de la primera versión del programa de las N Reinas contra la cantidad de reinas.

A partir de 10 de entrada, el paralelismo del programa rebasa la cantidad de núcleos de cualquier arquitectura disponible en el mercado.

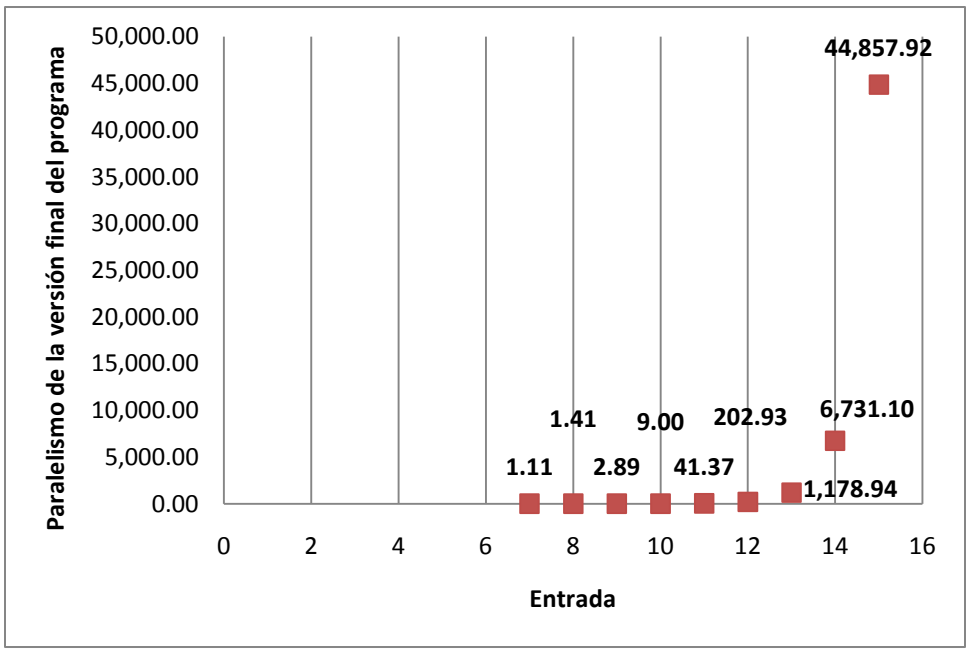

Figura 14. Gráfica del paralelismo de la versión final del programa de las N Reinas contra la cantidad de reinas.

Análisis de tiempo de ejecución

En este trabajo, el análisis de tiempo de ejecución de este algoritmo se realiza con 16 reinas. Esto debido a que el programa de las N reinas tiene muy alto paralelismo, y evaluar la función con 16 reinas permite

estudiar la mejora en desempeño independientemente de las limitaciones de paralelismo. Los resultados pueden permitir estudiar la fidelidad del modelo de trabajo y complejidad, y sus desventajas al ser aplicado en un caso real.

En el apéndice 1 es posible consultar todos los puntos que fueron revisados para el análisis de tiempo de ejecución.

El programa de las N reinas tiene una efectividad de **0.7319**. Se puede apreciar que la efectividad fue un poco más alta en este caso, respecto del programa del cálculo de la serie de Fibonacci. Las tareas del programa de las N Reinas tienen segmentos de código más amplios que los de la serie de Fibonacci, esto permite que la ejecución de tareas en paralelo amortice el costo de crearlas; es decir que entre más código del programa ejecute cada uno de los hilos, el costo de crear tareas paralelas se vuelve menos significativo.

En la figura 15, se muestran las gráficas de la aceleración del programa en función de la cantidad de hilos de ejecución que son creados para ejecutar el programa:

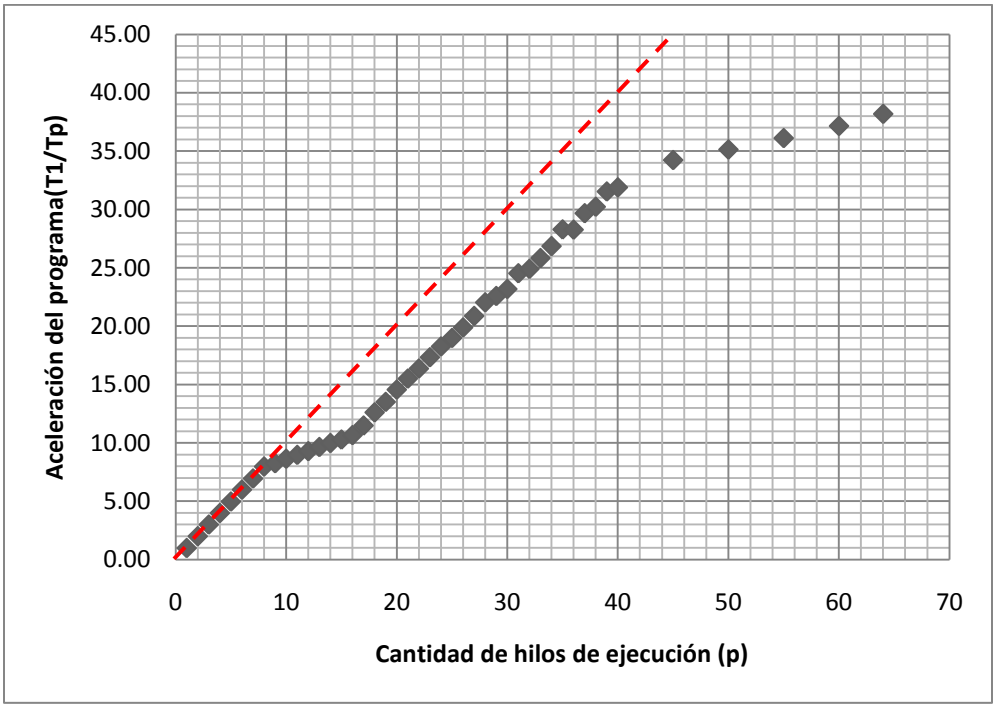

Figura 15. Gráfica de las muestras de aceleración del programa de las N Reinas

Es muy interesante observar qué similar es la figura 15, donde se grafica la aceleración del programa de las N Reinas, a la figura 12, donde se grafica la aceleración del programa e la serie de Fibonacci. Las gráficas tienen las mismas tendencias, y el análisis de los resultados es más o menos el mismo:

Al igual que la figura 12, para el análisis de los resultados de este programa, conviene dividir la gráfica en 4 partes, según el comportamiento de los tiempos de ejecución:

1. La sección de ecuación de tasa de desempeño lineal. Las muestras obtenidas para 1, 2, 3, 4, 5, 6, 7 y 8 hilos de ejecución cumplen con la ecuación de tasa de desempeño lineal. En la tabla del apéndice 1 se puede apreciar que todas estas muestras tienen una eficiencia prácticamente de 1; es decir que todos los núcleos en que se ejecuta el programa son utilizados adecuadamente. Al igual que en el algoritmo anterior, la ecuación tasa lineal de desempeño se mantiene.
2. La sección de muestras entre 9 y 16, donde la pendiente es menor a la de las dos secciones contiguas, y la eficiencia es muy mala. Al igual que en el problema de la serie de Fibonacci, esto es debido a la planificación de hilos dentro de los mismos núcleos compartiendo recursos con HyperThreading.
3. La sección media. Las muestras obtenidas a partir de 17 y hasta 45 hilos de ejecución tienen aceleración sublineal, pero pueden ser más o menos ajustadas a una recta con ordenada al origen negativa. Las eficiencias de esta sección están distribuidas con media **0.7666** y desviación estándar **0.05084**.
4. La sección final. Las muestras obtenidas para más de 45 hilos de ejecución tienen muy mala tasa de aceleración por núcleo, y eficiencias por debajo de **0.66**. Esto debido seguramente a todos los costos incurridos en la planificación, comunicación, creación de tareas, robo de trabajo y compartición de recursos.

3. Ordenamiento por mezcla

Análisis de escalabilidad

También para el ordenamiento por mezcla es importante observar el análisis de escalabilidad. Recordemos que para este algoritmo se hicieron dos implementaciones; una donde el paralelismo fue de orden $O(\log n)$ debido a la mezcla serial de arreglos y otra donde al paralelizar la mezcla, el paralelismo teórico aumentó a orden $O\left(\frac{n}{\log^2 n}\right)$.

	Paralelismo	
Entrada	**Primera versión**	**Versión final**
2048	3.38	6.97
4096	5.25	11.61
8192	7.95	19.99
16384	10.85	30.90
32768	12.67	42.95
65536	13.57	54.18
131072	13.93	63.13

Según el análisis teórico, la implementación con mezcla paralelizada tendrá mayor escalabilidad que aquella que realiza la mezcla en serie. A continuación se muestran los resultados del análisis de escalabilidad para ambas implementaciones:

En ambos casos se verifica la validez del modelado teórico; el paralelismo de ambos crece de manera sublineal según la entrada. Como era esperado, el paralelismo de la versión final del programa es mucho mayor que el paralelismo de la versión con mezcla serial.

La primera versión tiene un crecimiento que prácticamente se estanca, y aunque sigue creciendo indefinidamente con el tamaño del problema, su tasa de crecimiento es demasiado baja, y no se escala suficientemente.

La versión final del programa, con la función *merge* ya paralelizada, muestra una tasa de crecimiento mucho mejor, y en general da un mucho mejor paralelismo para cualquier caso. Aunque se comprueba que la función es de orden inferior al lineal, se escala en una tasa adecuada, y aunque lentamente, mantiene una tasa aceptable de crecimiento según el tamaño del problema. En la figura 16 se aprecia la comparación del paralelismo de ambas versiones:

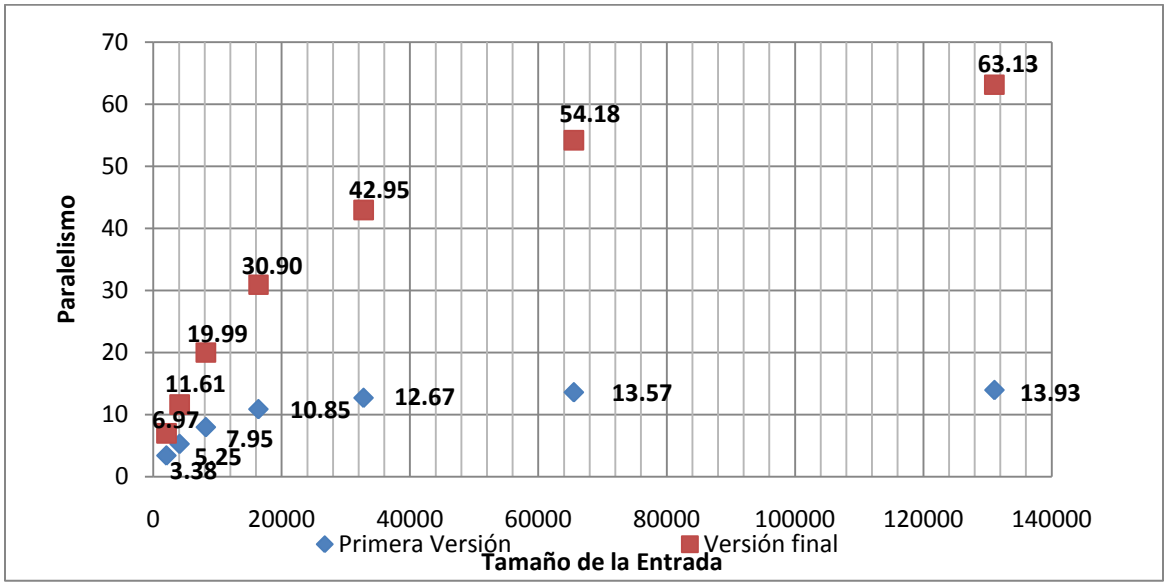

Figura 16. Gráfica de paralelismo de los programas de *mergesort* contra el tamaño de la entrada.

Como es posible apreciar de la figura 16, el paralelismo de la primera versión tiene una forma logarítmica, que a partir de 30 mil elementos a ordenar deja de escalarse. Por otro lado, la versión final del algoritmo tiene una forma sublineal, pero escala a tasas mucho mayores que la versión inicial.

Esto nos permite comprobar la hipótesis inicial:

i. Al paralelizar la función de mezcla, se obtiene un algoritmo mucho más escalable y con más paralelismo.

Es decir que una mezcla en serie limita el paralelismo del algoritmo, y ultimadamente limita su velocidad al ejecutarlo en arquitecturas altamente paralelas.

Análisis de tiempo de ejecución

Para este algoritmo, el análisis de tiempo de ejecución se ejecutó ordenando un arreglo de 4,194,304 elementos, para obtener la mayor cantidad de paralelismo disponible –que aún es mucho menor que el caso de las N Reinas y la serie de Fibonacci.

En el apéndice 1 se muestran los resultados de las pruebas de ejecución del programa de ordenamiento por mezcla.

La efectividad del programa es de **0.2883**; es decir que los costos incurridos por la creación de tareas paralelas y planificación de hilos son demasiado altos, y afectan la ejecución del programa, haciendo que la versión paralela ejecutada en un solo hilo tome casi 4 veces lo que toma ejecutar la versión serial. De hecho, el tiempo de ejecución de la versión paralela nunca pasa por debajo del tiempo de ejecución de la versión serial; sin importar la cantidad de hilos de ejecución que sean creados.

En la figura 17 se muestran los resultados del tiempo de ejecución del algoritmo en función de la cantidad de hilos de ejecución que se utilizaron para realizarlo. No tiene caso presentar una gráfica de aceleración del algoritmo, ya que en general no se presentó aceleración sustancial; los resultados son interesantes, y vale la pena hacer un análisis profundo:

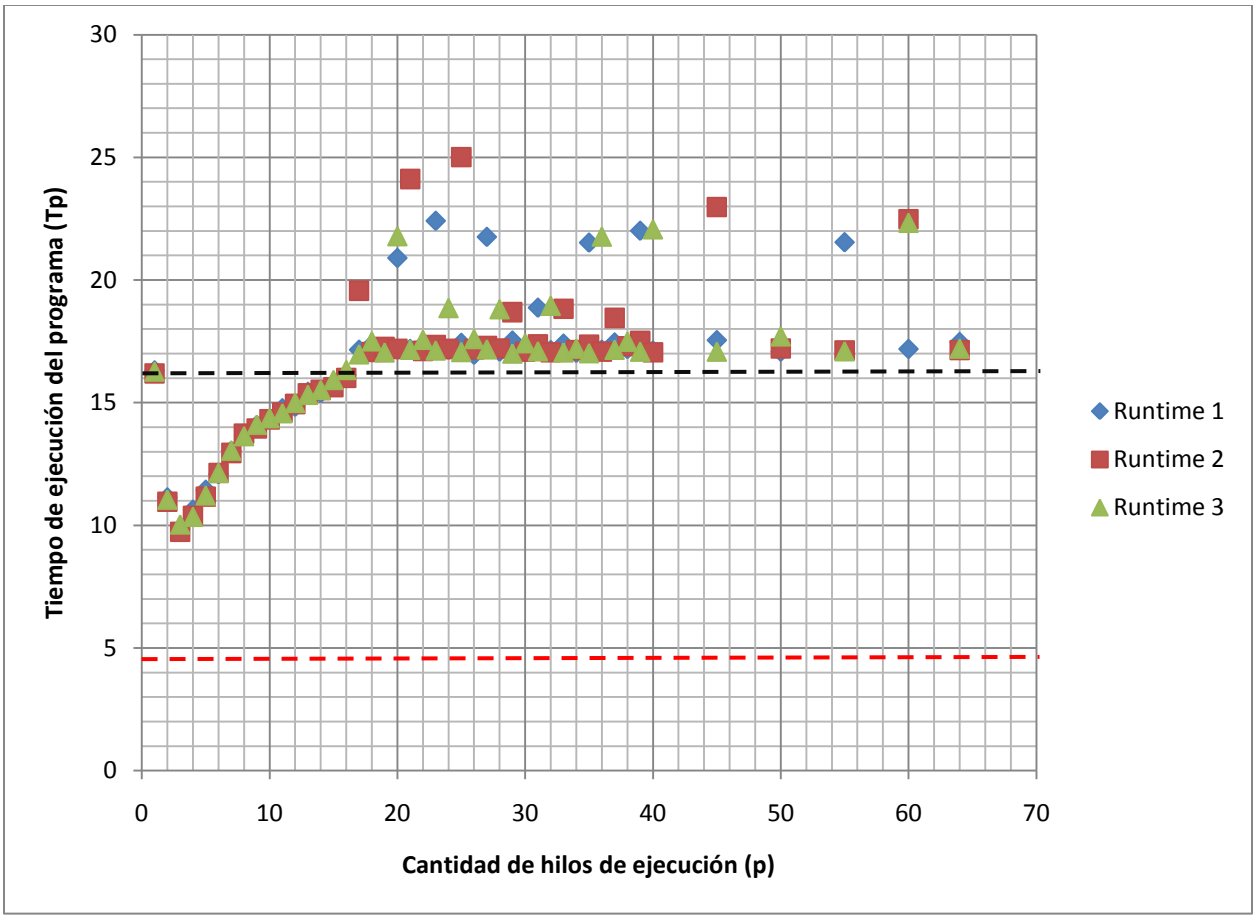

Figura 17. Tiempo de ejecución del ordenamiento por mezcla en tres pruebas distintas.

Como se observa en la gráfica, el tiempo de ejecución del algoritmo no disminuyó mucho; alcanzó su punto mínimo al ejecutarse con 3 hilos. La línea punteada obscura marca el tiempo con un solo hilo de ejecución, mientras que la línea punteada roja marca el tiempo de ejecución de la versión serial del programa. Como se puede apreciar de la gráfica, la versión paralela nunca logra ejecutarse en tiempo

menor que la versión serial. Para este algoritmo no valió la pena paralelizarlo, pues de hecho se empeora el desempeño respecto de la versión serial.

Uno de los motivos principales por los que el paralelismo que se calculó para el algoritmo no fue representativo y no permitió estimar realmente qué tanto se aceleraría el programa es que Cilk++ calcula el trabajo y la profundidad como la cantidad total de instrucciones en ensamblador que son ejecutadas y la cantidad máxima de instrucciones que deben ser ejecutadas en serie; luego divide estas dos cantidades y con esto calcula el paralelismo. El problema aquí es que Cilk++ no toma en cuenta en ningún momento los costos de intercomunicación de la arquitectura, sino que simplemente calcula el paralelismo y todas las cantidades en número de instrucciones de ensamblador. Esto significaría que cada instrucción es considerada de costo unitario; y esto no es cierto: Existen instrucciones más costosas, y especialmente las de intercomunicación tendrán costos de ejecución mayores.

Otro detalle interesante que observar es que para más de 16 hilos de ejecución, el programa se ejecuta más lento de lo que se ejecutaría en un solo núcleo, pero se mantiene alrededor de 17 segundos en tiempo de ejecución; además, también a partir de 16 hilos de ejecución, se observan lecturas inestables, donde el tiempo de ejecución se dispara en algunos casos y se mantiene estable en otros, sin alguna razón aparente.

En este caso es posible ver cómo la intercomunicación entre procesadores colocados en distintos chips se vuelve costosa para el algoritmo; y la planificación se vuelve más compleja, provocando que el tiempo de ejecución empeore y se vuelva inestable. Es decir que al tener que salir del mismo chip, el costo de planificación e intercomunicación daña el desempeño del programa, y la planificación no determinística hace que el tiempo de ejecución sea también no determinístico.

Por un lado, la falta en aceleración del algoritmo significa que no se contó con suficiente paralelismo; es decir que los costos de intercomunicación, planificación y robo de tareas entre los hilos de ejecución fueron demasiado altos, y el *parallel slack*, es decir, la diferencia entre la cantidad de unidades de procesamiento disponibles y el paralelismo de un programa, no fue suficientemente amplia. De manera que para verdaderamente obtener una mejora escalable del tiempo de ejecución de un algoritmo, es necesario contar con muchísimo paralelismo en el programa.

También es interesante observar que el comportamiento del tiempo de ejecución del programa cambia, al igual que en los programas anteriores, cuando la cantidad de hilos de ejecución creados hace que sea necesario planificar hilos en más de un chip; es decir, cuando es necesario que la comunicación salga del chip para crear y planificar hilos de ejecución en otro procesador.

4. El problema del vendedor viajero con búsqueda Tabu

El caso del problema del vendedor viajero es distinto de los algoritmos anteriores. La intención con este algoritmo no es acelerar su ejecución, sino utilizar los recursos paralelos disponibles para aumentar la cantidad de soluciones exploradas por el algoritmo.

Al generar varias tareas paralelas que se encargan de explorar soluciones en búsqueda de rutas cortas, cada tarea encuentra una solución cuya ruta es la mínima. Para este algoritmo, el estudio del tiempo de ejecución permite observar qué tan costosa es la creación y distribución de nuevas tareas, así como el paso de reducción.

Otra característica importante que observar del algoritmo es su uso de memoria. La implementación serial utiliza estructuras de datos bastante grandes, y la implementación paralela aumenta el uso de memoria linealmente con la cantidad de hilos de ejecución.

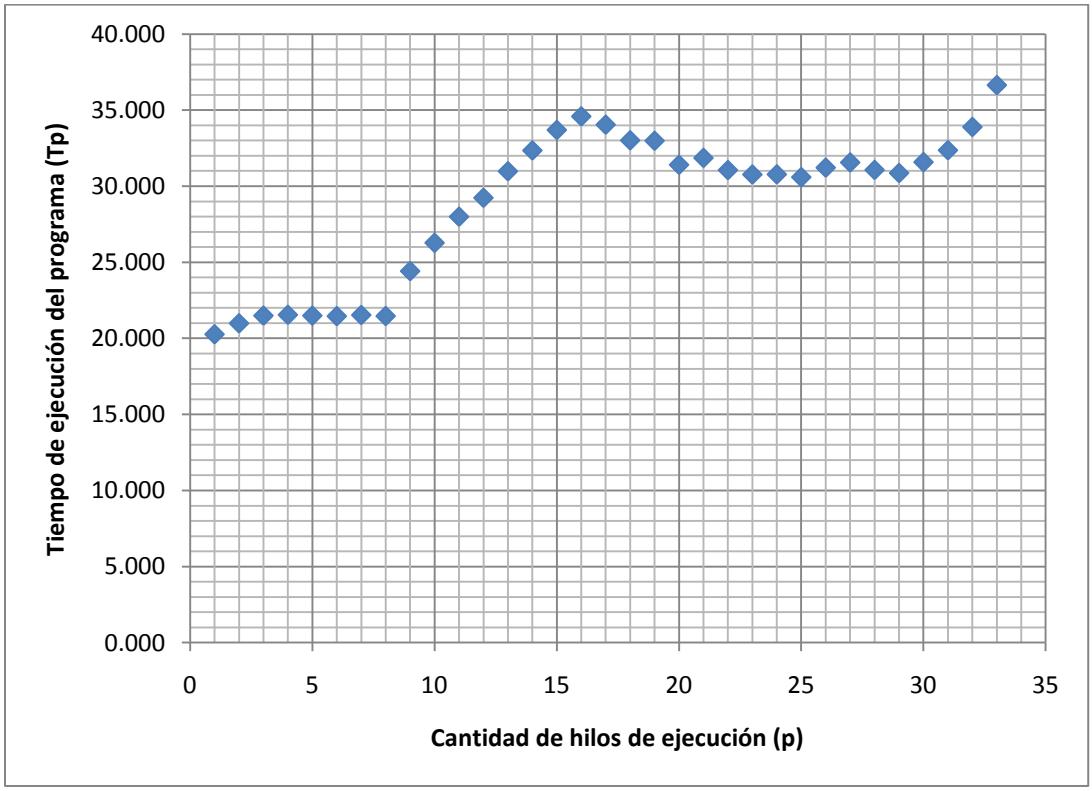

Figura 18. Gráfica del tiempo de ejecución del programa de búsqueda para el vendedor viajero.

Al igual que en los programas anteriores, parece ser que las capacidades de HyperThreading de los núcleos del procesador juegan un papel importante en el desempeño de la aplicación. Desde 1 hasta 8 hilos de ejecución, el desempeño del programa es más o menos constante, y se sitúa alrededor de los 22 segundos en tiempo de ejecución. Una vez que se utilizan más de 8 hilos de ejecución, y hasta 16 hilos de ejecución –justamente los que pueden ser planificados dentro de un mismo chip utilizando sus capacidades de HyperThreading-, la gráfica muestra claramente cómo el tiempo de ejecución se va degradando casi linealmente.

Al utilizar más de 16 hilos de ejecución, el tiempo de ejecución del programa vuelve a estabilizarse.

Para más de 33 hilos de ejecución, el programa fue siempre terminado por el sistema operativo antes de concluir. Esto debido seguramente al uso intensivo de memoria del programa. Para 33 hilos de ejecución, el proceso ***Tabu_search_parallel.exe*** llega a reservar hasta 2gb de memoria para su ejecución.

Referencias

[1]. Blumofe, R.D. Joerg, C.F. Kuszumalu, B.C. Leiserson, C. Randall, K.H. Zhou, Y. (Julio 1995). Cilk: An Efficient Multithreaded Runtime System. *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Capítulo 5. Conclusiones

Los resultados obtenidos de los experimentos han mostrado ciertos patrones interesantes. Por un lado, el modelo de trabajo-profundidad mostró ser adecuado, pero no perfecto; los resultados dependieron también, y en gran medida, de los detalles la arquitectura donde fueron ejecutados, y de costos ocultos, como la planificación y la intercomunicación en el sistema.

En todos los experimentos, un resultado claro e inesperado fue la manera en que la planificación de varios hilos por núcleo con HyperThreading se manifestó sobre el desempeño de las aplicaciones. Los programas con muchos cálculos independientes (N Reinas, Fibonacci y el vendedor viajero), es decir, con mucho paralelismo, fueron afectados negativamente por el HyperThreading, provocando que el desempeño por hilo disminuyera. Vale mencionar que el desempeño global del programa mejoró, pero la ecuación de tasa de desempeño lineal dejó de cumplir al empezar a planificar hilos con HyperThreading; es decir que al afectar el desempeño por hilo, el modelo de trabajo y profundidad fue incapaz de predecir precisamente el desempeño de la aplicación.

No existe mucha información disponible sobre el planificador de procesos de Windows; sin embargo, una técnica común de planificación utilizada por los sistemas operativos es planificar hilos por afinidad de memoria; esto significa que hilos del mismo proceso serán planificados dentro del mismo chip para aprovechar la afinidad y la localidad de sus accesos de memoria [1]. Por esto es que el sistema operativo procura planificar hilos en el mismo chip aún cuando esto signifique utilizar HyperThreading y que al tener dos hilos compartiendo los mismos recursos se pierda desempeño por hilo.

El programa del problema del vendedor viajero con búsqueda tabú también fue afectado al compartir recursos por la planificación por afinidad con HyperThreading. De hecho, éste es un ejemplo donde el planificar fuera del chip habría sido más efectivo que planificar por afinidad, ya que cada una de las tareas generadas utiliza datos completamente distintos. Esto significa que no sólo son afectados al compartir los recursos de ejecución del mismo núcleo (como predictores de ramas, ALUs, etc), sino que además se afecta el desempeño porque ambos hilos comparten todos los niveles en la jerarquía de memoria; es decir que deben compartir un solo cache para acceder a datos completamente diferentes.

Por otro lado, una de las ventajas de la planificación por afinidad es que los costos de intercomunicación y de planificación se hacen mínimos, además de ganarse tiempo dado que dos procesadores lógicos (es decir, los dos hilos que se ejecutan en un solo núcleo con HyperThreading), comparten los mismos niveles de memoria, y los problemas de coherencia en el caché, y la consecuente pérdida de desempeño que pueden ocasionar, no son algo que afecte el desempeño de los hilos. Un ejemplo muy claro de esto es el programa de ordenamiento por mezcla, que al planificar en el mismo chip, mantiene desempeño por encima del desempeño del programa en un solo núcleo, y al empezar a planificar fuera del mismo procesador, el desempeño cae, y el tiempo de ejecución se vuelve inestable.

Ahora, en entornos donde los costos de planificación, intercomunicación, coherencia de memoria y compartición de recursos se minimizan, el modelo de trabajo-profundidad fue muy apropiado al explicar el comportamiento de los algoritmos. Los dos ejemplos más claros son el algoritmo de las N Reinas y de

la serie de Fibonacci, que al ejecutarse con 1 a 8 hilos de ejecución, (justamente la cantidad de hilos de ejecución que pueden ser planificados dentro de un solo chip sin utilizar HyperThreading), su tasa de desempeño respecto de la cantidad de núcleos disponibles fue constante.

Para los algoritmos que tienen mucho paralelismo, existe una gran cantidad de cálculos independientes. Esto significa que cada hilo utiliza ampliamente los recursos del núcleo al que es asignado; esto explica que al compartir el mismo núcleo con otro hilo, el desempeño disminuyera.

Respecto del modelo trabajo-profundidad, el algoritmo de ordenamiento por mezcla permite tomar una lección importante: A pesar de contar con bastante paralelismo en el programa, no fue suficiente para mejorar el tiempo de ejecución; es decir que para paralelizar un algoritmo o un programa es necesario buscar tasas muy altas de paralelismo o no valdrá la pena el esfuerzo. En este caso sólo se realizó esfuerzo y se perdió en desempeño.

El principal motivo por el que el paralelismo calculado por Cilk++ no fuera acorde con la aceleración que experimentó el programa fue seguramente que no se tomaron en cuenta los costes de intercomunicación; y que Cilk++ utiliza únicamente la cuenta de instrucciones para calcular el paralelismo. Sabemos que especialmente para las arquitecturas x86, el costo de cada instrucción es distinto, y también lo es con la intercomunicación entre los núcleos de los procesadores; es decir que la manera en que Cilk++ calcula el paralelismo de una aplicación es consistente únicamente cuando existen costes de intercomunicación muy pequeños.

Otra consideración importante de la programación paralela que se manifestó en uno de los programas es el uso de memoria. El programa del vendedor viajero fue el caso más notorio del exceso en uso de memoria, y en este caso se convirtió en un problema, al grado de que no es posible ejecutarlo con más de 33 hilos, o la memoria que utiliza es demasiada; en los otros programas, el uso de memoria no fue ningún problema; sin embargo, no se debe tomar a la ligera; en cualquier programa con N hilos, se tienen N pilas separadas guardando variables locales, más la memoria que cada hilo localice dinámicamente. Cilk++ procura disminuir el uso de memoria, localizando memoria únicamente cuando es estrictamente necesario [2]; sin embargo, un hilo en ejecución reserva siempre espacio de memoria; y es importante no perder de vista esto al desarrollar algoritmos y aplicaciones que exploten paralelismo, o aún más, que realicen tareas independientes.

En el campo de las plataformas de programación, se puede esperar que los modelos, los lenguajes y las APIs de programación paralela empiecen a enfocarse en la productividad del programador; que será sin duda el enfoque principal, moviendo otros aspectos importantes como el desempeño al segundo plano. Si una plataforma no puede habilitar a un programador a generar código correcto y eficiente con buena productividad, esta plataforma se condena a sí misma al abandono y al desinterés de la ingeniería de software. Herramientas como lenguajes paralelos simplificados y compiladores más sofisticados, así como tal vez, herramientas de compilación y optimización en tiempo de ejecución, se convertirán en protagonistas de la revolución paralela. [3]

Por otro lado, los arquitectos de hardware están en una difícil encrucijada, pero al igual que las plataformas de programación, el diseño de hardware será llevado por el camino de la productividad y la

facilidad de programación. Esto dejaría fuera arquitecturas heterogéneas que explotan características de las cargas de trabajo, pero dificultan el trabajo del programador. Aunque esto es una aserción un tanto prematura, el camino de las microarquitecturas de Intel puede servirnos como referencia y como evidencia. Las arquitecturas homogéneas parecen ser la opción a seguir, especialmente para el cómputo de propósito general.

Finalmente, como observación final, sólo queda decir que las consideraciones para el desarrollo de programas y algoritmos paralelos de alto desempeño son muchas; desde el paralelismo teórico disponible en el algoritmo, el rediseño de estructuras de datos y rediseño de algoritmos, así como aspectos prácticos, como la arquitectura en que es ejecutado, la eficiencia de la intercomunicación, la jerarquía de memoria, y otras tecnologías disponibles en la arquitectura; también aspectos del sistema operativo, como el planificador, la ligereza y agilidad de los hilos y los procesos que ejecuta, las características del lenguaje en que se desarrolla, etc. Es un proceso complejo y elaborado, pero que, cada vez más, se mueve hacia los reflectores y gana importancia en el terreno del cómputo y la ingeniería de software.

Me gustaría concluir invitando al departamento de ingeniería en computación a incorporar temas relacionados con el cómputo paralelo en el contenido curricular de la carrera. El liderazgo en esta área es obligatorio para una institución académica de la talla y la importancia de la Facultad de Ingeniería de la UNAM.

Referencias

[1]. Nakajima, J. Pallipadi, V.
(2002). Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling.
WIESS '02 Conference on Industrial Experiences with Systems Software.

[2]. Blumofe, R.D. Joerg, C.F. Kuszumalu, B.C. Leiserson, C. Randall, K.H. Zhou, Y.
(Julio 1995). Cilk: An Efficient Multithreaded Runtime System.
5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[3]. Hwu, W. Keutzer, K. Mattson, T.G.
(Julio 2008). The Concurrency Challenge.
IEEE Design and Test of Computers, 25, 312-320.

Apéndice 1. Tablas de resultados de los algoritmos

El cálculo de los números de Fibonacci

Cálculo de la serie de Fibonacci - 45 - Versión paralela						
Tserial: 125.7591						Efectividad: 0.6888
Hilos de ejecución	Ejecución 1	Ejecución 2	Error	Tiempo promedio (Tp)	Aceleración (T1/Tp)	Eficiencia (T1/(Tp*p))
1	182.5665	181.0227	0.84561	181.795	1.00	1.0000
2	91.6311	90.7297	0.983727	91.180	1.99	0.9969
3	60.6841	60.7777	0.154241	60.731	2.99	0.9978
4	45.9136	45.8484	0.142006	45.881	3.96	0.9906
5	37.0291	36.7196	0.835829	36.874	4.93	0.9860
6	30.8927	30.5292	1.176653	30.711	5.92	0.9866
7	26.4475	26.1768	1.023537	26.312	6.91	0.9870
8	23.2453	23.4	0.665511	23.323	7.79	0.9743
9	22.3852	22.2456	0.623626	22.315	8.15	0.9052
10	21.4312	21.4344	0.014932	21.433	8.48	0.8482
11	20.7272	20.5764	0.727546	20.652	8.80	0.8003
12	19.9368	19.9836	0.234742	19.960	9.11	0.7590
13	19.2504	19.422	0.89141	19.336	9.40	0.7232
14	18.7356	18.6732	0.333056	18.704	9.72	0.6942
15	17.9712	18.018	0.260417	17.995	10.10	0.6735
16	17.5968	17.55	0.265957	17.573	10.34	0.6466
17	16.2396	16.0992	0.864553	16.169	11.24	0.6614
18	14.8356	14.7264	0.736067	14.781	12.30	0.6833
19	13.7748	13.572	1.472254	13.673	13.30	0.6998
20	12.8232	12.8544	0.243309	12.839	14.16	0.7080
21	11.9964	12.0276	0.260078	12.012	15.13	0.7207
22	11.3568	11.31	0.412088	11.333	16.04	0.7291
23	10.7016	10.764	0.58309	10.733	16.94	0.7364
24	10.2492	10.0776	1.674277	10.163	17.89	0.7453
25	9.6408	9.6876	0.485437	9.664	18.81	0.7524
26	9.1728	9.0948	0.85034	9.134	19.90	0.7655
27	8.6892	8.6424	0.5386	8.666	20.98	0.7770
28	8.4084	8.3616	0.556586	8.385	21.68	0.7743
29	8.1432	8.0808	0.766284	8.112	22.41	0.7728
30	7.8156	7.7688	0.598802	7.792	23.33	0.7777
31	7.4256	7.4256	0	7.426	24.48	0.7897
32	7.098	7.176	1.098901	7.137	25.47	0.7960

Cálculo de la serie de Fibonacci - 45 - Versión paralela						
Hilos de ejecución	**Ejecución 1**	**Ejecución 2**	**Error**	**Tiempo promedio (Tp)**	**Aceleración (T1/Tp)**	**Eficiencia (T1/(Tp*p))**
33	7.0044	6.942	0.890869	6.973	26.07	0.7900
34	6.6144	6.552	0.943396	6.583	27.61	0.8122
35	6.318	6.318	0	6.318	28.77	0.8221
36	6.3336	6.318	0.246305	6.326	28.74	0.7983
37	6.1308	6.1776	0.763359	6.154	29.54	0.7984
38	5.9904	5.9748	0.260417	5.983	30.39	0.7997
39	5.8032	5.694	1.88172	5.749	31.62	0.8109
40	5.6472	5.6472	0	5.647	32.19	0.8048
45	5.4132	5.46	0.864553	5.437	33.44	0.7431
50	5.226	5.226	0	5.226	34.79	0.6957
55	5.0232	5.0076	0.310559	5.015	36.25	0.6590
60	4.8048	4.836	0.649351	4.820	37.71	0.6286
64	4.68	4.7268	1	4.703	38.65	0.6039

El problema de las N Reinas

N Queens - 16 Reinas - Versión Final						
Tserial: 258.602						Efectividad: 0.7319
Hilos de ejecución	Ejecución 1	Ejecución 2	Error	Tiempo promedio (Tp)	Aceleración (T1/Tp)	Eficiencia (T1/(Tp*p))
1	354.155	352.467	0.477	353.311	1.00	1.0000
2	175.672	175.672	0.000	175.672	2.01	1.0080
3	117.577	117.343	0.199	117.46	3.01	1.0040
4	88.659	88.0778	0.656	88.3684	4.00	0.9986
5	71.301	70.6213	0.953	70.96115	4.98	0.9934
6	58.858	58.8901	0.055	58.87405	6.00	1.0029
7	50.7313	50.8405	0.215	50.7859	6.96	0.9973
8	44.2729	44.2729	0.000	44.2729	7.98	0.9999
9	42.8221	42.6973	0.291	42.7597	8.26	0.9189
10	40.8877	40.8253	0.153	40.8565	8.65	0.8662
11	39.3745	39.3433	0.079	39.3589	8.98	0.8177
12	38.0173	38.0017	0.041	38.0095	9.30	0.7763
13	36.5821	36.6133	0.085	36.5977	9.65	0.7447
14	35.4121	35.2873	0.352	35.3497	9.99	0.7144
15	34.2421	34.4137	0.501	34.3279	10.29	0.6895
16	33.1501	33.0565	0.282	33.1033	10.67	0.6677
17	30.7477	30.7321	0.051	30.7399	11.49	0.6775
18	28.002	27.9864	0.056	27.9942	12.62	0.7026
19	26.13	26.1456	0.060	26.1378	13.52	0.7133
20	24.2424	24.258	0.064	24.2502	14.57	0.7304
21	22.7604	22.7916	0.137	22.776	15.51	0.7410
22	21.5748	21.5748	0.000	21.5748	16.38	0.7461
23	20.3892	20.358	0.153	20.3736	17.34	0.7552
24	19.4376	19.2504	0.963	19.344	18.26	0.7592
25	18.6576	18.5484	0.585	18.603	18.99	0.7593
26	17.7528	17.7684	0.088	17.7606	19.89	0.7673
27	17.0664	16.77	1.737	16.9182	20.88	0.7686
28	15.9432	16.1148	1.076	16.029	22.04	0.7933
29	15.6468	15.6312	0.100	15.639	22.59	0.7805
30	15.3504	15.1164	1.524	15.2334	23.19	0.7690
31	14.352	14.4456	0.652	14.3988	24.54	0.7960
32	14.2896	14.0712	1.528	14.1804	24.92	0.7745
33	13.7592	13.5876	1.247	13.6734	25.84	0.7800
34	13.26	13.0416	1.647	13.1508	26.87	0.7855
35	12.5424	12.4332	0.871	12.4878	28.29	0.8068

N Queens - 16 Reinas - Versión Final						
Hilos de ejecución	**Ejecución 1**	**Ejecución 2**	**Error**	**Tiempo promedio (Tp)**	**Aceleración (T1/Tp)**	**Eficiencia (T1/(Tp*p))**
36	12.5112	12.48	0.249	12.4956	28.27	0.7863
37	11.778	12.0276	2.119	11.9028	29.68	0.8127
38	11.6532	11.7156	0.535	11.6844	30.24	0.7998
39	11.3256	11.076	2.204	11.2008	31.54	0.8018
40	11.1072	11.0448	0.562	11.076	31.90	0.7971
45	10.4364	10.2024	2.242	10.3194	34.24	0.7541
50	9.9684	10.14	1.721	10.0542	35.14	0.7106
55	9.8124	9.75	0.636	9.7812	36.12	0.6562
60	9.60962	9.4068	2.111	9.50821	37.16	0.6142
64	9.29762	9.204	1.007	9.25081	38.19	0.5952

El algoritmo de ordenamiento por mezcla

Mergesort - 4194304 elementos - Versión final							
Tserial: 4.6882							Efectividad: 0.2883
Hilos de ejecución	**Ejecución 1**	**Ejecución 2**	**Ejecución 3**	**Error**	**Tiempo promedio(Tp)**	**Aceleración (T1/Tp)**	**Eficiencia (T1/(Tp*p))**
1	16.3176	16.1949	16.2691	0.349725	16.2605	1.000	1.0000
2	11.1384	10.9592	11.0448	0.816395	11.0475	1.472	0.7359
3	9.8436	9.7344	10.0308	0.264131	9.8696	1.648	0.5492
4	10.6392	10.3896	10.3428	1.710655	10.4572	1.555	0.3887
5	11.4504	11.1696	11.2164	1.498638	11.2788	1.442	0.2883
6	12.09	12.1368	12.1368	0.258065	12.1212	1.341	0.2236
7	13.0416	12.948	13.0416	0.239234	13.0104	1.250	0.1785
8	13.6656	13.7436	13.6344	0.114155	13.6812	1.189	0.1486
9	14.0868	13.9464	14.1024	0.295312	14.0452	1.158	0.1286
10	14.3208	14.3208	14.3676	0.108932	14.3364	1.134	0.1134
11	14.7732	14.6016	14.5704	0.844773	14.6484	1.110	0.1009
12	14.8356	14.9448	14.9916	0.595864	14.9240	1.090	0.0908
13	15.444	15.3816	15.3192	0.40404	15.3816	1.057	0.0813
14	15.3972	15.522	15.522	0.540358	15.4804	1.050	0.0750
15	15.7092	15.6312	15.9276	0.297915	15.7560	1.032	0.0688
16	16.146	16.0056	16.3332	0.096618	16.1616	1.006	0.0629
17	17.16	19.5624	16.9728	4.30303	17.8984	0.908	0.0534
18	17.3628	17.0664	17.5032	0.299491	17.3108	0.939	0.0522
19	17.0508	17.2692	17.0508	0.426959	17.1236	0.950	0.0500
20	20.904	17.1912	21.7932	4.502488	19.9628	0.815	0.0407
21	17.1912	24.1176	17.16	13.36963	19.4896	0.834	0.0397
22	17.3628	17.1132	17.5656	0.089847	17.3472	0.937	0.0426
23	22.4172	17.3472	17.1288	15.40246	18.9644	0.857	0.0373
24	17.16	17.1912	18.8604	3.363636	17.7372	0.917	0.0382
25	17.4408	25.0068	17.0664	13.74478	19.8380	0.820	0.0328
26	16.9572	17.1756	17.5968	1.686599	17.2432	0.943	0.0363
27	21.762	17.316	17.1756	13.83513	18.7512	0.867	0.0321
28	17.082	17.2068	18.8136	3.622527	17.7008	0.919	0.0328
29	17.5344	18.688	17.004	1.184719	17.7421	0.916	0.0316
30	17.0976	17.082	17.4252	0.608273	17.2016	0.945	0.0315
31	18.876	17.3784	17.0976	5.785124	17.7840	0.914	0.0295
32	17.1444	17.0508	18.954	3.336366	17.7164	0.918	0.0287
33	17.4096	18.8292	17.0508	2.031063	17.7632	0.915	0.0277
34	17.0352	17.1444	17.2068	0.549451	17.1288	0.949	0.0279
35	21.528	17.3628	17.0196	13.42995	18.6368	0.872	0.0249

Mergesort - 4194304 elementos - Versión final							
Hilos de ejecución	**Ejecución 1**	**Ejecución 2**	**Ejecución 3**	**Error**	**Tiempo promedio(Tp)**	**Aceleración (T1/Tp)**	**Eficiencia (T1/(Tp*p))**
36	17.16	17.082	21.7776	8.818182	18.6732	0.871	0.0242
37	17.4564	18.4548	17.16	1.340483	17.6904	0.919	0.0248
38	17.1912	17.3004	17.4876	0.786449	17.3264	0.938	0.0247
39	22.0116	17.5188	17.082	14.26884	18.8708	0.862	0.0221
40	17.1288	17.0508	22.074	9.471767	18.7512	0.867	0.0217
45	17.55	22.9788	17.082	9.422222	19.2036	0.847	0.0188
50	17.082	17.2068	17.706	1.461187	17.3316	0.938	0.0188
55	21.5436	17.1288	17.0976	13.70987	18.5900	0.875	0.0159
60	17.1912	22.4796	22.3392	20.23593	20.6700	0.787	0.0131
64	17.472	17.144	17.2068	1.131716	17.2743	0.941	0.0147

El algoritmo del vendedor viajero con búsqueda tabú

Tabu search - 100 ciud. - V. final						
Tserial: 20.051						
Hilos de ejecución	**Tiempo 1**	**Tiempo 2**	**Tiempo 3**	**Error**	**Tiempo promedio(Tp)**	**Aceleración (T1/Tp)**
1	20.141	20.0772	20.592	0.640816	20.270	1.0000
2	21.0242	20.9351	21.0132	0.158706	20.991	0.9657
3	21.6922	21.2341	21.5592	0.908314	21.495	0.9430
4	21.8563	21.4032	21.3876	1.405849	21.549	0.9406
5	21.1042	21.1692	22.2144	1.856187	21.496	0.9430
6	21.1562	21.3876	21.84	1.441973	21.461	0.9445
7	21.6172	21.5124	21.4812	0.371309	21.537	0.9412
8	21.4732	21.5904	21.3408	0.023595	21.468	0.9442
9	24.5303	24.5388	24.1956	0.443261	24.422	0.8300
10	25.7244	25.896	27.222	2.162927	26.281	0.7713
11	28.8601	27.456	27.6744	2.991212	27.997	0.7240
12	29.2813	29.6245	28.8133	0.14207	29.240	0.6932
13	31.6057	30.6385	30.7009	1.974327	30.982	0.6543
14	32.7133	31.8085	32.5261	1.112697	32.349	0.6266
15	33.6961	33.9301	33.4621	0	33.696	0.6016
16	34.6321	34.9753	34.1641	0.12012	34.591	0.5860
17	34.1173	33.8053	34.2109	0.213381	34.045	0.5954
18	33.1345	32.5261	33.3841	0.360953	33.015	0.6140
19	33.5869	32.7913	32.5729	1.795938	32.984	0.6145
20	31.4809	31.6681	31.0753	0.231251	31.408	0.6454
21	31.3561	31.9801	32.2453	1.608618	31.861	0.6362
22	31.2937	30.6853	31.2001	0.747754	31.060	0.6526
23	30.8413	30.7477	30.7165	0.236047	30.769	0.6588
24	30.9817	30.9661	30.3889	0.65458	30.779	0.6586
25	30.2797	30.7321	30.7789	1.047567	30.597	0.6625
26	31.4809	30.4357	31.7461	0.825898	31.221	0.6492
27	31.3405	31.7901	31.5589	0.710476	31.563	0.6422
28	31.3717	31.4497	30.3889	0.961376	31.070	0.6524
29	29.8429	31.3405	31.4185	3.432642	30.867	0.6567
30	32.0737	30.0301	32.6509	1.52399	31.585	0.6418
31	32.5105	32.2141	32.3701	0.447855	32.365	0.6263
32	34.1017	34.5541	33.0253	0.60994	33.894	0.5980
33	35.5525	36.9409	37.4401	3.071514	36.645	0.5532
34-64	-	-	-		-	-