VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

POSGRADO EN CIENCIA E INGENIERIA DE LA COMPUTACION

"ANALISIS DEL DESEMPEÑO POR LA MIGRACION DE THREADS EN UNA ARQUITECTURA MULTI-CORE"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A

RAMSES LOPEZ GUERRERO

DIRECTOR DE TESIS:

JORGE LUIS ORTEGA ARJONA

Mexico, D.F. **2011**

Reconocimientos

Primordialmente agradezco a Dios por el apoyo y fortaleza brindados y por quien si no fuera por él no podría salir adelante. Este trabajo jamás se hubiera realizado sin tu ayuda.

Con gran estimación agradezco al M. en C. Iván Cervantes por su amistad y apoyo en todo momento. Este logro lo comparto con tigo.

Con enorme gratitud al Dr. Jorge Ortega, a quien admiro y siempre agradeceré por la enseñanza y guía en el camino del conocimiento. Este trabajo no hubiera sido posible sin tu dirección y seguimiento.

Agradecimientos

A Dios.

Al Dr. Héctor Benítez por su apoyo invaluable que me brindó durante mi estancia en su laboratorio.

Al Dr. Jesús Savage por su apoyo incondicional y como ejemplo de motivación personal durante mi estancia en el posgrado.

Al M. en C. Marcelo Pérez por su apoyo invaluable como persona, amigo y académico.

A María Sánchez por su tolerancia y ayuda invaluable al mostrarme que el mundo se puede modelar en objetos.

Al Ing. Héctor Felix por su compañerismo y amistad sincera durante la estancia en el posgrado.

Al L. en C.C. Carlos Zerón por su compañerismo, amistad y apoyo incondicional durante mi estancia en el posgrado.

Dedicatoria

A Dios.

A mis padres, Teresa Guerrero y Víctor López por su apoyo en todo momento.

A mis hermanos, Miguel López y Víctor López por ser un ejemplo a seguir y por su apoyo incondicional.

A mi gran amiga Veronica Jaramillo, por su sincera amistad y su poyo incondicional en todo momento, a Thierry Busca, por su apoyo brindado y a quien admiro como persona, y a mis nobles amigos, Andrés Montiel y Norberto Espinoza. Gracias a todos por su incondicionable amistad.

Por último, pero no por ello menos importante, a Jonathan Ponciano quien siempre ha sido un invaluable amigo. Gracias por tu cariño, consejo y apoyo brindado.

Con todos ellos comparto mi logro.

Índice general

1. Introducción **7**
- 1.1. El contexto . . . 7
- 1.2. El problema . . . 8
- 1.3. La hipótesis . . . 9
- 1.4. El enfoque . . . 10
- 1.5. Contribuciones . . . 10
- 1.6. Estructura de la tesis . . . 11

2. Antecedentes **14**
- 2.1. Una introducción a multi-core . . . 14
  - 2.1.1. Ventajas . . . 15
  - 2.1.2. Desventajas . . . 16
  - 2.1.3. Aplicaciones . . . 16
- 2.2. Arquitectura multi-core . . . 17
  - 2.2.1. Estructura física . . . 17
  - 2.2.2. Comunicación multi-core . . . 20
- 2.3. Procesamiento multi-core . . . 22
  - 2.3.1. Una introducción al procesamiento multi-core . . . 22
  - 2.3.2. Memoria caché . . . 24
  - 2.3.3. Multi-Procesamiento simétrico . . . 26
  - 2.3.4. Multi-Procesamiento asimétrico . . . 26
  - 2.3.5. Multi-Procesamiento limitado . . . 27
- 2.4. Medidas de desempeño . . . 28
  - 2.4.1. Tiempo de ejecución . . . 28
  - 2.4.2. Aceleración . . . 29
  - 2.4.3. Eficiencia . . . 29
  - 2.4.4. Costo . . . 29
- 2.5. Resumen . . . 30

3. Trabajo relacionado **31**
3.1. Costo de la migración de threads 31
3.1.1. Objetivo 32
3.1.2. Experimentación 32
3.1.3. Resultados 34
3.1.4. Conclusiones 34
3.1.5. Contribuciones 34
3.2. Implicaciones en el desempeño en la migración de un thread . 35
3.2.1. Objetivo 35
3.2.2. Experimentación 35
3.2.3. Resultados 37
3.2.4. Conclusiones 42
3.2.5. Contribuciones 43
3.3. Resumen 43

4. Análisis del desempeño por la migración de threads **45**
4.1. Migración de thread 45
4.1.1. Fases en la migración de un thread 46
4.1.2. Otros factores en la migración de un thread 48
4.2. Arquitectura multi-core 49
4.3. OpenMP 50
4.3.1. Interfaz de afinidad de thread de OpenMP de Intel . . 51
4.4. Programa 53
4.4.1. Producto de una matriz por un vector 53
4.4.2. Algoritmo paralelo y desbalanceado 54
4.4.3. Código 55
4.5. Medición del desempeño 61
4.6. Resumen 62

5. Resultados experimentales **63**
5.1. Mediciones de las pruebas 63
5.1.1. Política 1 63
5.1.2. Política 2 65
5.1.3. Política 3 67
5.1.4. Política 4 69
5.2. Resultados experimentales 71
5.2.1. Política 1 72
5.2.2. Política 2 73
5.2.3. Política 3 74
5.2.4. Política 4 75

5.3. Resumen 75

6. Conclusiones **77**
6.1. Retomando la hipótesis 77
6.1.1. Discusión 77
6.1.2. Interpretación y análisis de los resultados 79
6.2. Comparación con el trabajo relacionado 81
6.3. Trabajo a futuro 83

Índice de figuras

2.1. CPU con un chip single-core 18
2.2. CPU con un chip multi-core 19
2.3. a) Arquitectura multi-core con memorias caché privadas de nivel 1. b) Arquitectura multi-core con memorias caché privadas de nivel 1 de datos y de instrucciones y una memoria caché compartida de nivel 2. c) Arquitectura multi-core con memorias caché privadas de nivel 1 de datos y de instrucciones, memorias caché privadas de nivel 2 y una memoria caché compartida de nivel 3 19
2.4. Jerarquía de la memoria 21
2.5. Proceso lógico o espacio de direcciones virtual 22
2.6. Múltiples threads en un proceso 23
2.7. Ciclo de vida de un thread 24

3.1. Arquitectura multi-core 33
3.2. Arquitecturas multi-core 36

4.1. Diferentes fases en la migración de un thread desde el core C1 hacia el core C2 47
4.2. Diagrama de la arquitectura multi-core 50
4.3. Bifuración-Unión. Modelo de programación que soporta OpenMP 51
4.4. Programa paralelo. Procesamiento simultáneo que se realiza por un conjunto de threads para obtener el producto de la matriz por el vector 55
4.5. Programa desbalanceado. Procesamiento simultáneo y desbalanceado que se realiza por un conjunto de threads para obtener el producto de la matriz por el vector 55

5.1. Política 1 72
5.2. Política 2 73

5.3. Política 3 74
5.4. Política 4 75

Índice de tablas

5.1. Escenario de experimentación para 800 threads en la política 1 64
5.2. Escenario de experimentación para 400 threads en la política 1 64
5.3. Escenario de experimentación para 200 threads en la política 1 65
5.4. Escenario de experimentación para 100 threads en la política 1 65
5.5. Escenario de experimentación para 800 threads en la política 2 66
5.6. Escenario de experimentación para 400 threads en la política 2 66
5.7. Escenario de experimentación para 200 threads en la política 2 67
5.8. Escenario de experimentación para 100 threads en la política 2 67
5.9. Escenario de experimentación para 800 threads en la política 3 68
5.10. Escenario de experimentación para 400 threads en la política 3 68
5.11. Escenario de experimentación para 200 threads en la política 3 69
5.12. Escenario de experimentación para 100 threads en la política 3 69
5.13. Escenario de experimentación para 800 threads en la política 4 70
5.14. Escenario de experimentación para 400 threads en la política 4 70
5.15. Escenario de experimentación para 200 threads en la política 4 71
5.16. Escenario de experimentación para 100 threads en la política 4 71
5.17. Tiempos de ejecución promedio para la política 1 72
5.18. Tiempos de ejecución promedio para la política 2 73
5.19. Tiempos de ejecución promedio para la política 3 74
5.20. Tiempos de ejecución promedio para la política 4 75

6.1. Análisis de resultados experimentales de la política 1 80
6.2. Análisis de resultados experimentales de la política 2 80
6.3. Análisis de resultados experimentales de la política 3 81
6.4. Análisis de resultados experimentales de la política 4 81

Capítulo 1

Introducción

1.1. El contexto

La industria del cómputo está impulsada por la búsqueda de un cada vez mayor desempeño. Los clientes esperan obtener más rápidos, eficientes y poderosos equipos de cómputo que van desde equipos de propósito específico, como los hay en telecomunicaciones, redes y aviónica, hasta los equipos embebidos de baja potencia, como en videojuegos, computadoras de escritorio y equipos portátiles [9].

El cómputo con procesadores multi-core surge entonces. La tecnología en los procesadores ha alcanzado dimensiones por debajo de los 45 nm (nanómetros – 1×10^{-9}) es decir, procesadores dual-core y quad-core ya han salido al mercado, pero las dimensiones alcanzadas pueden ser utilizadas para diseñar procesadores para decenas de cores. El enfoque multi-core puede ser denominado ulteriormente como many-core o masivamente multi-core.

Este nivel de cómputo ha provocado un nuevo reto para la industria semiconductora, además de los diseñadores de sistemas y de software. El paralelismo es el reto esencial en el procesamiento multi-core, pues la ejecución de aplicaciones en paralelo es difícil lograrlo de una manera eficiente y rápida.

El paralelismo se puede analizar en cuatro formas generales [8, 9]:

- **Paralelismo a nivel de bit.-** Se basa en incrementar la medida de la palabra en una computadora.
- **Paralelismo a nivel de instrucción.-** Es la técnica para identificar instrucciones que no dependen una de la otra, tales como trabajar con diferentes variables y ejecutarlas al mismo tiempo.

- **Paralelismo de datos.-** Permite a múltiples unidades de procesamiento ejecutar los datos simultáneamente.

- **Paralelismo de tareas.-** Distribuye diferentes aplicaciones, procesos o threads a diferentes unidades de procesamiento. Esto puede ser hecho manualmente o con la ayuda del sistema operativo.

1.2. El problema

Es importante mencionar que el desempeño es el objetivo principal al utilizar sistemas paralelos [28, 29] y la paralelización es el reto esencial en el desarrollo de todo sistema multi-core [9].

El paralelismo en sistemas multi-core incrementa la dificultad en el análisis del desempeño por la inclusión de un nuevo factor: *la migración de threads entre los cores.* La migración puede ocurrir por las políticas del balanceo de carga en el sistema operativo y se refiere a la decisión de qué core va a ejecutar qué thread. En arquitecturas single-core, es decir, arquitecturas donde el procesador se constituye de un core, cuando la multi-programación surge, el sistema operativo se encarga de calendarizar y ejecutar los procesos. Con esto se logra un mejor desempeño en el sistema; cuando el multi-threading surge, el sistema operativo se encarga de calendarizar y ejecutar los threads, con lo cual se logra un mejor desempeño en el sistema. Ahora bien, en arquitecturas multi-core, cuando la migración de threads entre los cores surge, se desconoce si el sistema obtiene un mejor desempeño con un costo aceptable cuando se realiza el balanceo de carga en el sistema operativo entre los cores.

Para aprovechar los procesadores multi-core se espera obtener un alto grado de paralelismo de threads para obtener un mejor desempeño, pero esta circunstancia puede coexistir con los siguientes escenarios:

- La decisión y proceso de migrar threads entre los cores consume tiempo de procesamiento; entonces podría suceder que la ejecución de los threads sea más rápida sin realizar la migración.

- Cuando se migra un thread entre dos cores se pueden tener pérdidas de datos; entonces si existen excesos de migración, se tienen pérdidas innecesarias de datos [37, 43].

- Si el algoritmo o los datos son secuenciales por naturaleza, o bien, cuando existe una pesada y bien integrada tarea, surgen dificultades en la distribución; por ejemplo en algoritmos de cifrado. El Triple DES

(Triple Data Encryption Standard) y el AES (Advanced Encryption Standard) son difíciles paralelizar por su naturaleza secuencial [43].

Sin embargo, en otros escenarios la paralelización es más sencilla, se pueden mencionar los siguientes:

- Para sistemas donde existan pequeñas unidades o que no existan tareas fuertemente relacionadas, es más fácil paralelizar los threads [9].
- Si el algoritmo o los datos son paralelos por naturaleza ó casi-paralelos, la migración de thread es más sencilla porque los datos están interrelacionados en menor grado y con ello existen pocas o no existen referencias a localidades de memoria donde los threads compartan recursos; de tal manera que si no existen threads que compartan datos y/o instrucciones, no se requiere de procesamiento para enlazar datos e/o instrucciones a los threads.

La migración de threads entre cores es una tarea que consume tiempo. El balanceo de carga puede ser ineficiente para la carga paralela de trabajo, y en el peor de los casos, puede contribuir a la degradación en el desempeño del sistema.

1.3. La hipótesis

El principal objetivo de esta tesis se puede expresar con la confirmación de la siguiente hipótesis:

> "Dado un programa paralelo y desbalanceado en una arquitectura multi-core, ¿se puede obtener un mejor desempeño en la ejecución del programa debido a no realizar migración de threads entre los cores?"

Se debe establecer lo que significa *"un programa paralelo y desbalanceado"* y lo que significa *"obtener un mejor desempeño"*. Se proponen las siguientes dos suposiciones:

1. *Programa paralelo.* Es un conjunto de instrucciones que en tiempo de ejecución las operaciones son realizadas simultáneamente por más de un procesador ó core.
2. *Programa desbalanceado.* Es un conjunto de instrucciones que en tiempo de ejecución la cantidad de trabajo no es uniformemente distribuida a través de todos los procesadores ó cores en el sistema.

Se define lo que significa *"obtener un mejor desempeño"* como sigue: un *programa paralelo* y *desbalaneado* sin la migración de threads entre los cores tiene un mejor desempeño si entrega tiempos de ejecución menores que si fueran realizados con la migración de threads entre los procesadores ó cores.

1.4. El enfoque

El objetivo primordial de este trabajo es obtener la información que influya en la decisión de saber si la migración de threads es benéfica en el desempeño de la ejecución de un programa paralelo y desbalanceado en una arquitectura multi-core.

El método implica la ejecución de un programa constituido por algoritmo y datos. El algoritmo del programa no contiene una relación entre los datos de los threads a migrar, el tamaño de los datos de los threads es uniforme y la ejecución del procesamiento paralelo es menor a 55 milisegundos; además, la arquitectura a utilizar, es una arquitectura simétrica en todos los cores y en todos los niveles jerárquicos de memoria; entonces, se propone analizar el tiempo de ejecución como la medida de desempeño a utilizar. Esta labor se reproduce diez veces para obtener medidas de desempeño suficientes y tener confiabilidad en el resultado de los métodos estadísticos.

De tal manera, se propone en los dos siguientes escenarios:

1. Analizar el programa paralelo y desbalanceado en escenarios con migración de threads.

2. Analizar el programa paralelo y desbalanceado en escenarios sin migración de threads.

Entonces, los resultados de los métodos estadísticos muestran el desempeño en los escenarios. Estos resultados permiten conocer cuál es el mejor escenario para la ejecución de un programa paralelo y desbalanceado en una arquitectura multi-core.

1.5. Contribuciones

La meta de esta tesis es analizar el desempeño de un programa paralelo y desbalanceado en dos escenarios de ejecución. El objetivo es conocer si la migración de threads beneficia al desempeño de sistemas multi-core. Esta

aportación permite a los diseñadores en una arquitectura multi-core construir mejores sistemas al tener una visión del impacto en el desempeño en la migración de threads.

Si la migración de threads aporta un beneficio insignificante o si es perjudicial al desempeño del sistema en comparación a un esquema que no contenga migración de threads, se beneficia en:

- Disminuir la carga a los procesadores al eliminar la migración de threads.
- Disminuir la dificultad de la programación en la comunicación intercores al eliminar la migración de threads.
- Disminuir el consumo de energía en los cores.

1.6. Estructura de la tesis

Esta tesis se estructura de la siguiente manera:

- **Capítulo 2: Antecedentes** Este capítulo presenta una introducción a la tecnología multi-core. Con el objetivo de clarificar esta tecnología, se detalla dónde se puede utilizar, las aplicaciones que se favorecen, sus ventajas e inconvenientes y se menciona como se puede beneficiar al explotar el paralelismo a nivel de threads.

  También se detalla la comunicación y estructura física de los microprocesadores, se menciona la estructura jerárquica de memoria y se detalla la estructura y características del nivel de memoria caché.

  Por otro lado, se presenta una introducción al procesamiento multicore, se menciona el procesamiento de un sistema operativo multitareas y se explica el flujo de control; se describe y detalla el procesamiento en la memoria caché y se explican los tipos de procesamiento que existen en la ejecución paralela de un programa.

  Para finalizar, se mencionan las características en la evaluación del desempeño de un programa que se ejecuta en paralelo y se especifican algunas medidas del desempeño.

- **Capítulo 3: Trabajo relacionado** Este capítulo presenta una revisión del trabajo relacionado en el desempeño en la migración de threads.

Se analizan dos trabajos relevantes. El primero se denomina "Entendimiento del Costo de la Migración de Threads para Aplicaciones Java Multi-Threaded que se ejecutan sobre una plataforma Multi-Core" (*Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform*) y el segundo se denomina "Implicaciones en el Desempeño en la Migración de un Thread sobre un Chip Multi-Core" (*Performance Implications of Single Thread Migration on a Chip Multi-Core*).

El trabajo relacionado se organiza de la siguiente manera: se presenta una introducción al trabajo relacionado; se enfatiza en el problema, los objetivos y en la importancia de la investigación. Posteriormente se describe el planteamiento del escenario de experimentación y se indica el desarrollo y parámetros a evaluar. Luego se presentan los resultados y con ello las conclusiones. Al finalizar, se mencionan las contribuciones del trabajo en estudio.

- **Capítulo 4: Análisis del desempeño por la migración de threads** El objetivo de este capítulo es analizar la migración de threads, describir el escenario de experimentación y especificar la medición del desempeño para realizar la experimentación del trabajo de investigación.

  Con el propósito de comprender el proceso y *overhead* en la migración de un thread, se define y detalla el proceso que se requiere para cambiar el flujo de control entre dos cores de procesamiento.

  Con la finalidad de detallar el escenario de experimentación y los requisitos necesarios para poder realizar los escenarios de experimentación, con y sin la migración de threads, se especifica la arquitectura, el programa y los parámetros a utilizar en la investigación: la arquitectura multi-core, el microprocesador, el sistema operativo, el kernel, el compilador y las directivas a utilizar, la interfaz de afinidad de thread y el procesamiento multi-core.

  Por último y con el objetivo de poder realizar la medición de desempeño, se especifica cómo se presenta la medición del tiempo del programa paralelo y desbalanceado en los dos escenarios de migración.

- **Capítulo 5: Resultados experimentales** Este capítulo presenta los escenarios de experimentación y los resultados de las mediciones del trabajo de investigación. Con el propósito de obtener la información que influya en la decisión de saber si la migración de threads beneficia

al desempeño en la ejecución de un programa paralelo y desbalanceado en la arquitectura multi-core, se plantean cuatro políticas que rigen la realización de los experimentos. Para finalizar se muestran los resultados experimentales.

- **Capítulo 6: Conclusiones** Este capítulo expone una evaluación del trabajo de investigación presentado en forma de un resumen crítico, y se retoma la hipótesis y contribución de la tesis para su justificación, se presentan las aportaciones de la investigación y se plantean nuevas vías de estudio.

  Con el objetivo de justificar la hipótesis y contribuciones de la tesis, se presenta una recapitulación crítica en la sección de interpretación y análisis de resultados. El análisis de los resultados detalla los resultados experimentales que se obtienen de los escenarios de ejecución.

  Con la finalidad de mencionar las aportaciones de este trabajo de investigación, se realiza una comparación con el trabajo relacionado que se cree se acerca más a este trabajo de tesis.

  Para finalizar, se plantean nuevos objetivos e identifican nuevas vías de investigación como trabajo a futuro.

Capítulo 2

Antecedentes

Este capítulo presenta una introducción a la tecnología multi-core. Con el objetivo de clarificar esta tecnología, se detalla dónde se puede utilizar, las aplicaciones que se favorecen, sus ventajas e inconvenientes y se menciona como se puede beneficiar al explotar el paralelismo a nivel de threads.

También se detalla la comunicación y estructura física de los microprocesadores, se menciona la estructura jerárquica de memoria y se detalla la estructura y características del nivel de memoria caché.

Por otro lado, se presenta una introducción al procesamiento multi-core, se menciona el procesamiento de un sistema operativo multi-tareas y se explica el flujo de control; se describe y detalla el procesamiento en la memoria caché y se explican los tipos de procesamiento que existen en la ejecución paralela de un programa.

Para finalizar, se mencionan las características en la evaluación del desempeño de un programa que se ejecuta en paralelo y se especifican algunas medidas del desempeño.

2.1. Una introducción a multi-core

El cómputo paralelo puede incrementar la velocidad, eficiencia y desempeño de las computadoras si se colocan dos o más cores en el mismo chip con el propósito de ejecutar tareas simultáneas. A dicho planteamiento se le denomina *multi-core* o *tecnología multi-core* [22] (Figura 2.2).

De manera detallada, las arquitecturas de procesadores multi-core contienen dos o más cores de ejecución o motores de cálculo (*computational engines*) en un solo procesador. El procesador se conecta directamente dentro de un socket de procesador y el sistema operativo percibe a cada uno

de los cores de ejecución como procesadores lógicos discretos y a todos los recursos asociados de ejecución [15, 35].

Cuando la tecnología multi-core se analiza en comparación con la tecnología single-core, se puede citar lo siguiente [14]:

2.1.1. Ventajas

El beneficio del desempeño en una arquitectura single-core tiene limitantes. Éstos dependen de la aceleración de la frecuencia del reloj [24]:

- **Barrera de energía.-** El consumo de energía aumenta al acelerar la frecuencia.
- **Barrera de memoria.-** En una arquitectura single-core, la velocidad entre el procesador y la memoria es cada vez más desproporcional. Un cuello de botella en el desempeño puede ser causado por el acceso a la memoria principal. Esta peculiaridad puede encauzar al incremento de memoria caché para enmascarar la latencia; sin embargo, si es excesivo el incremento de memoria caché, el desempeño puede verse afectado.
- **Barrera del paralelismo a nivel de instrucción.-** En un procesador single-core se necesita realizar paralelismo en el flujo único de instrucciones (*single instructions stream*), pero depender del desempeño por el paralelismo de instrucciones en un cada vez más concurrido core es difícil.

El desempeño en multi-core se obtiene con el mismo o con menos consumo de energía y con una disminución de temperatura en el sistema a diferencia de un procesador single-core [24].

- Un procesador multi-core cuenta con frecuencia de reloj menor (menor velocidad) y puede ejecutar más operaciones por segundo; con lo que se obtiene ahorro de energía.
- Desde que los cores se encuentran en el mismo chip en una tecnología multi-core, las señales que viajan entre éstos recorren distancias más cortas, con lo que se obtiene rapidez y ahorro de energía.
- Si en multi-core las señales que viajan entre los cores recorren distancias cortas, las señales se degradan menos, lo que provoca que las señales no se repitan con frecuencia. Por lo que a menor repetición de señales, se ahorra más energía.

En multi-core se aprovecha mejor el espacio comparado a los procesadores single-core debido a que los múltiples cores sobre un circuito integrado se colocan en el mismo procesador. Incluso pueden compartir interfaz de bus y circuitos de memoria caché. Por lo tanto, los circuitos para la coherencia de la memoria caché son rápidos ya que operan en el mismo circuito integrado (ver Sección 2.2).

2.1.2. Desventajas

En arquitecturas multi-core se requiere adaptar el sistema operativo y las aplicaciones que se ejecutan sobre ésta, para acrecentar la utilización de los recursos de cómputo. Además, el desempeño exige que las aplicaciones implementen verdadero multi-threading [24] (ver Sección 2.1.3).

Por otro lado, si en una arquitectura multi-core, los cores comparten el bus del sistema y el ancho de banda de la memoria, se puede limitar al desempeño; por ejemplo, si un procesador single-core se limita por el ancho de banda de la memoria, una arquitectura dual-core con el mismo problema puede obtener una mejora del 30 % y hasta el 70 %. Si el ancho de banda de la memoria no es problema, se puede obtener una mejora del 90 % [24].

Además, los chips multi-core son difíciles para administrar térmicamente.

2.1.3. Aplicaciones

La tecnología multi-core se puede usar en computadoras de escritorio, computadoras personales móviles, servidores, estaciones de trabajo, sistemas embebidos y redes de computadoras; es idónea en aplicaciones que demanden consumo de cómputo; por ejemplo, en aplicaciones científicas, CAD/CAM[1], procesamiento multimedia[2], procesamiento digital de señales, reconocimiento de patrones, gráficos, juegos en 3D, codificación, conversión de formatos de archivos en grabación de datos, aplicaciones de audio, datos, voz, video, radio y control, aplicaciones en telecomunicaciones como en 3G, WiMAX y celulares, en enrutamiento de paquetes (*packet routing*), en telefonía de paquetes, control de tráfico, filtrado, búsqueda de archivos, en buscadores web y en interfaces de usuario [17, 22, 24, 34].

Los sistemas multi-core, al igual que los sistemas single-core, pueden implementar arquitecturas escalares, VLIW, procesamiento vectorial, SIMD o multi-threading [24]. Para el usuario promedio, la mejora en el desempeño

[1] Diseño Asistido por Computadora/Fabricación Asistida por Computadora.

[2] Cualquier objeto o sistema que utiliza medios de expresión para presentar o comunicar información [25].

se percibe en el ambiente multi-tareas; por ejemplo, cuando en un procesador multi-core se observa una película y al mismo tiempo se realiza una detección de virus al sistema; ésto se debe a que cada aplicación se asigna hacia un core diferente [14].

El desempeño en procesadores multi-core depende en parte de los algoritmos y de la implementación del software; por ejemplo, si un usuario ejecuta una aplicación en una máquina multi-core, puede desaprovechar la tecnología por no utilizar verdadero multi-threading en las aplicaciones. Por ejemplo, un detector de virus puede correr en un thread mientras que la interfaz de usuario gráfica puede correr en otro thread. Esto no es un verdadero ambiente multi-threading. Un verdadero ambiente multi-threading es cuando la mayor parte del trabajo se divide en threads. El problema en el ejemplo anterior es que la interfaz de usuario gráfica hace un trabajo, mientras que el detector de virus realiza otro trabajo que no se puede dividir entre diferentes cores.

El desarrollo de aplicaciones en una arquitectura multi-core se debe enfocar en implementar modelos con verdadero multi-threading para beneficiar al desempeño de sistemas multi-core. Por ejemplo, si las aplicaciones se implementan sin multi-threading y se cuenta con un sistema operativo multi-threading, éste puede distribuir las tareas hacia los cores, en vez de distribuir threads.

2.2. Arquitectura multi-core

2.2.1. Estructura física

La tecnología multi-core puede variar en su estructura y componentes en [35]:

- El número de cores que están en un procesador.
- La distancia de un core a otro core en el procesador.
- La distancia de los cores a la(s) memoria(s).
- El número de cores que comparten memoria en un procesador.
- El número de cores que se comunican por memoria distribuida.
- El número de memorias que están en el procesador.
- El número de niveles, la jerarquía, el tamaño y el número de memorias caché que están en el procesador.

A continuación se explican los componentes alrededor de una arquitectura multi-core:

1. **Core.-** Parte del procesador que ejecuta la lectura y ejecución de las instrucciones [24]. Cada core tiene sus propios recursos: registros, unidades de ejecución, niveles de memoria caché, etc. [35] (Figura 2.1).

2. **Memoria caché.-** Áreas temporales de retención de datos [5] (Figura 2.3).

3. **Circuito integrado (CI) o chip [30].-** Dispositivo que combina docenas a millones de transistores en un bloque de material semiconductor denominado *die* (Figura 2.1).

4. **Multi-Chip.-** Múltiples circuitos integrados que se empaquetan juntos [14] y donde cada chip consiste en un die.

5. **Unidad central del procesador, CPU o procesador [30].-** Parte de la computadora que contiene la ruta de datos y de control, que contiene a los cores de ejecución [24] (Figura 2.1 y Figura 2.2).

6. **Paquete de procesador.-** El paquete del procesador (*processor package*) contiene dos o más cores de ejecución y es la parte que se puede ver y sostener con la mano; se compone de sustrato, con pines de metal o pads en la parte inferior y de manera usual un disipador de calor en la parte superior [5, 35].

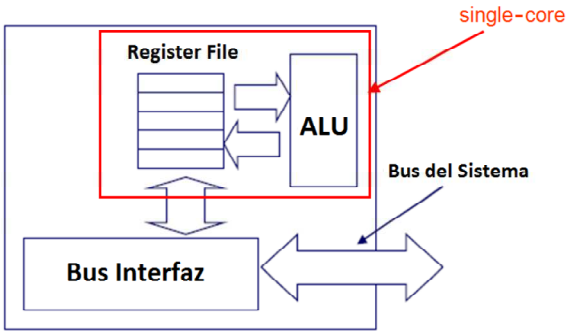

Figura 2.1: CPU con un chip single-core

En un paquete de procesador se entiende que a mayor integración de lógica exista, mayor recursos se comparten entre los cores sobre el die [35].

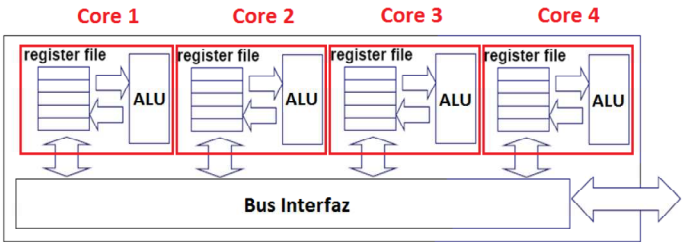

Figura 2.2: CPU con un chip multi-core

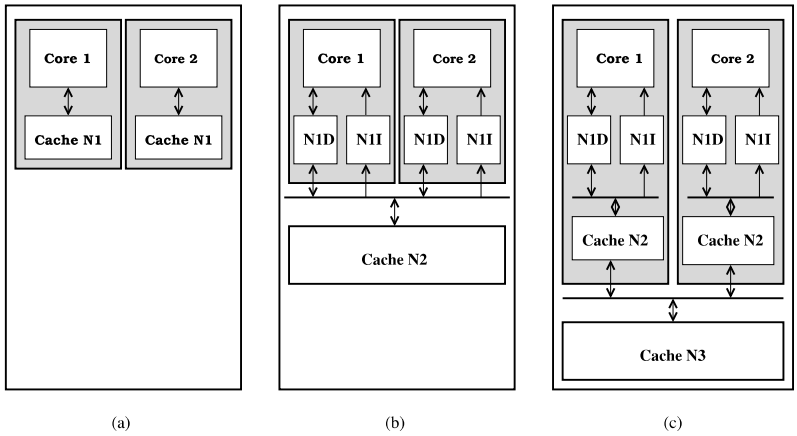

Figura 2.3: a) Arquitectura multi-core con memorias caché privadas de nivel 1. b) Arquitectura multi-core con memorias caché privadas de nivel 1 de datos y de instrucciones y una memoria caché compartida de nivel 2. c) Arquitectura multi-core con memorias caché privadas de nivel 1 de datos y de instrucciones, memorias caché privadas de nivel 2 y una memoria caché compartida de nivel 3

2.2.2. Comunicación multi-core

Una característica a tomar en cuenta de los procesadores multi-core es la manera en que los cores se comunican unos con otros y con la memoria. Los arquitectos de procesadores multi-core pueden diseñar a los cores para tener conexiones débiles o fuertes. Pueden, por ejemplo, implementar comunicación inter-core por memoria distribuida, memoria compartida o memoria compartida y distribuida. En procesadores many-core o masivamente multi-core (donde el número de cores es grande), nuevas técnicas de conexión entre los cores tienen que ser analizadas como conexiones de red sobre el chip. Topologías de red para inter-conectar cores incluyen bus, anillo, malla de dos dimensiones, jerárquica y crossbar [17, 24].

La comunicación sobre el chip y entre los chips determina el tiempo en que los cores tienen que esperar para acceder a los datos. Cuando los cores se encuentran en el mismo chip, las señales que viajan entre éstos recorren distancias más cortas, con lo que se obtiene mayor rapidez en la comunicación, y por tanto, los tipos de arquitecturas de comunicación dentro-del-chip y fuera-del-chip tienen impacto en la latencia del sistema [34].

Jerarquía de la memoria

Una jerarquía de memoria es una estructura que se constituye de niveles de memorias de diferentes medidas y velocidades. Cuanto más cerca se encuentre la memoria al microprocesador, la memoria será más pequeña y el tiempo de acceso será más rápido.

La jerarquía de la memoria puede consistir de múltiples niveles; sin embargo, los datos e instrucciones se transfieren entre dos niveles adyacentes de memoria. El nivel más cercano al procesador se denomina nivel jerárquico superior, es la memoria más pequeña y veloz de los niveles inferiores consecuentes.

El objetivo de diseñar una memoria como una jerarquía de niveles es presentar una memoria vasta (nivel jerárquico inferior), que sea accedida de manera rápida (nivel jerárquico superior) [30] (Figura 2.4).

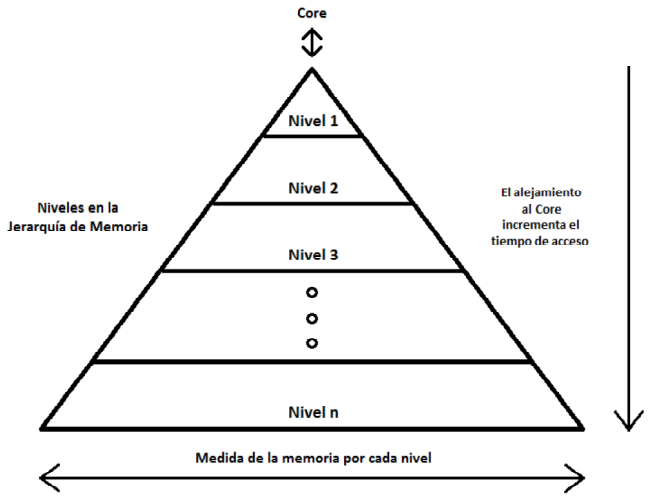

Figura 2.4: Jerarquía de la memoria

- **Memoria caché**

  Una memoria caché se estructura en *bloques o líneas de memoria caché*. Cada línea o bloque puede almacenar a la unidad mínima de información empleada en las memorias caché.

  De manera general, los arquitectos pueden diseñar a los procesadores multi-core con memoria caché privada y/o memoria caché compartida [2] (Figura 2.3).

  Ventajas con memoria caché privada:

  - Una memoria caché privada es más cercana al core, y por tanto más rápida.
  - Se reduce la contención.

  Ventajas con memoria caché compartida:

  - Los threads en diferentes cores pueden compartir datos.
  - Si existen pocos threads en el sistema, mayor espacio es disponible en la memoria caché.

2.3. Procesamiento multi-core

2.3.1. Una introducción al procesamiento multi-core

La idea en la implementación de la arquitectura interna del chip es la estrategia de *divide y vencerás*. Es decir, si el trabajo computacional que se realiza en un core de microprocesador en tecnologías single-core, es repartido sobre múltiples cores de ejecución (tecnología multi-core), entonces en un ciclo de reloj dado se puede ejecutar más trabajo computacional, y por tanto, se obtienen mejores tiempos de ejecución.

En un sistema operativo multi-tareas, el trabajo se divide en porciones semi-independientes denominadas tareas. Éstas se calendarizan e intercambian por medio del procesador y de las políticas del calendarizador para su ejecución. En diversos sistemas operativos multi-tarea, el trabajo se divide en procesos y también se puede dividir en threads [40].

Un proceso es el código de un programa que está en algún estado de ejecución y que tiene su propio espacio de direcciones; el sistema operativo se encarga de mapearlo, en partes o íntegro, hacia la memoria principal, y puede hacer uso de un espacio de memoria virtual. El proceso contiene un registro contador del programa que mantiene la dirección de la siguiente instrucción a ser ejecutada. Diversas implementaciones de sistemas operativos contienen al espacio de direcciones lógico del proceso estructurado de la siguiente forma: al inicio, en las direcciones bajas se encuentran las instrucciones o el código del programa que usualmente es acceso de sólo lectura, los datos globales del programa siguen, sean inicializados y/o no inicializados, después el montículo, en el cual se realiza la asignación de memoria dinámica que crece hacia las direcciones altas, en el fin de las direcciones altas del espacio de dirección del proceso y creciendo hacia la parte baja del fin de la pila del proceso se encuentra el almacenamiento para las variables locales de procedimientos, paso de parámetros y direcciones de regreso. El proceso tiene en este contexto un *thread* o *flujo de control* (ver Figura 2.5) [12].

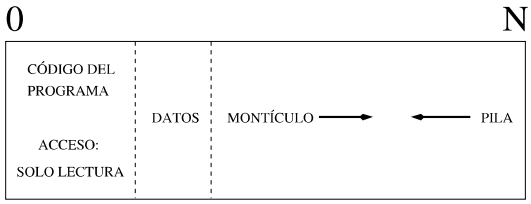

Figura 2.5: Proceso lógico o espacio de direcciones virtual

Un proceso puede solicitar al sistema operativo más de un flujo de control o thread en su espacio de direcciones. Los sistemas operativos que soportan esta característica se denominan *multi-thread*. Todos los threads en un proceso comparten el mismo espacio de direcciones del proceso (texto, datos, montículo, etc.) [35] y variables globales. Sin embargo, cada thread tiene su contador del programa, otros valores de registro de la CPU y tiempo de ejecución de la pila para llamadas a procedimientos [12] (Figura 2.6).

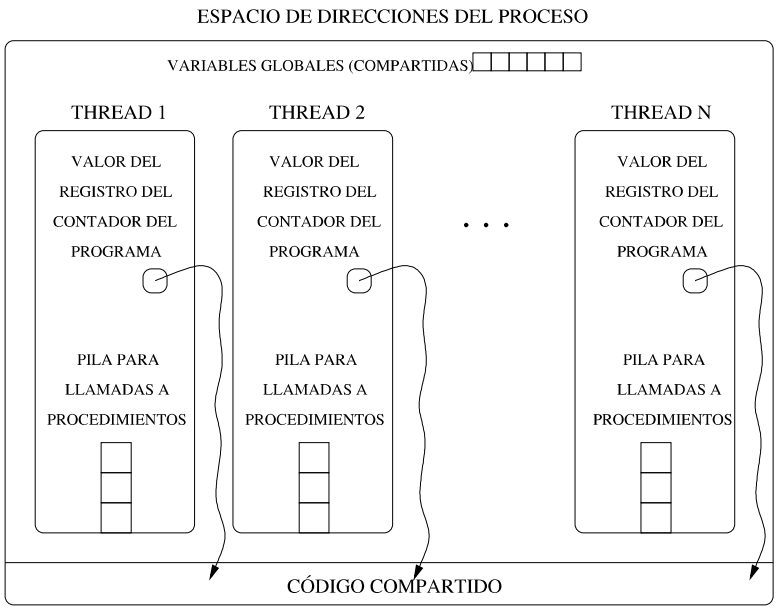

Figura 2.6: Múltiples threads en un proceso

Los estados de un thread son [21] (Figura 2.7)):

- **Listo.-** El thread está dispuesto a ejecutarse, pero espera en ser calendarizado para realizar procesamiento.
- **Corriendo.-** El thread se está ejecutando en un core.
- **Bloqueado.-** El thread puede no ejecutarse por esperar el acceso de un recurso compartido.
- **Terminado.-** El thread ha finalizado o se ha cancelado su ejecución. Los recursos del sistema son parcialmente liberados pero no completamente limpiados (*cleaned up*), la memoria de los threads es obsoleta pero puede retornar todavía valor.

- **(Reciclado).-** Esta fase la controla el sistema operativo y es donde los recursos del sistema son recuperados completamente.

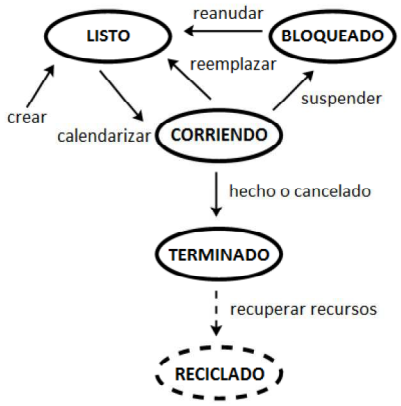

Figura 2.7: Ciclo de vida de un thread

2.3.2. Memoria caché

La cercanía del procesador o core de procesador con la memoria caché acelera el procesamiento de las instrucciones. Cuando existe un proceso, la memoria caché acumula datos e/o instrucciones que pueden ser valores duplicados de otro almacenamiento y/o valores operados. A la cantidad de información acumulada se le denomina afinidad de la memoria caché, y ésta puede ser alta o baja, dependiendo de la cantidad de información acumulada [18].

Los recursos compartidos entre los cores, como la memoria caché compartida, benefician la utilización de los recursos y la comunicación inter-cores. Por ejemplo, si varios threads se ejecutan sobre diferentes cores, los threads pueden compartir datos en la memoria caché compartida y disminuir la duplicación de datos por el último nivel de memoria caché [35].

Cuando el procesador solicita leer un dato o una instrucción y si ese dato o instrucción se encuentra en el bloque de memoria caché, se obtiene un *acierto* (*hit*). El controlador de la memoria caché se encarga de leer el contenido de la memoria caché y de enviar la información al procesador; de lo contrario, si el dato o instrucción no se encuentra en ese nivel de memoria, se obtiene un *desacierto* (*miss*), y el dato o instrucción, o se vuelve a calcular, o se busca en un nivel de memoria inferior.

Los desaciertos se clasifican en tres clases [30]:

- **Desaciertos de arranque en frío o desaciertos obligatorios.-** Éstos ocurren siempre que se accede a un bloque de memoria caché por primera vez. Se debe a que no existen instrucciones o datos cargados en memoria caché.

- **Desaciertos por capacidad.-** Éstos ocurren siempre que la memoria caché no puede contener todos los bloques necesarios durante la ejecución de un programa. Los bloques se reemplazan y se recuperan posteriormente.

- **Desaciertos por conflicto o desaciertos por colisión.-** Éstos ocurren en una memoria caché de conjunto-asociativo (*set-associative cache*) o en una memoria caché de correspondencia directa (*direct-mapped cache*) cuando múltiples bloques compiten por el mismo conjunto y se eliminan en una completamente asociativa memoria caché de la misma medida.

  En una memoria caché de conjunto-asociativo, hay un número de ubicaciones fijo (por lo menos dos) donde cada uno de los bloques se puede colocar. Una memoria caché de conjunto-asociativo con n ubicaciones para un bloque se denomina una memoria caché de conjunto-asociativo de n-bloques (*n-way set-associative*). Cada memoria caché de conjunto-asociativo de n-bloques consiste de un número de conjuntos, cada uno de los cuales consiste de n bloques.

  En una memoria caché de correspondencia directa cada ubicación de memoria le corresponde una ubicación de la memoria caché.

Cuando el procesador requiere escribir un dato o instrucción a memoria caché, el controlador de la memoria caché escribe ese dato o instrucción a la memoria caché; sin embargo, la memoria caché también se debe encargar de manipular las escrituras entre la memoria caché y el siguiente nivel inferior de memoria para que los datos o instrucciones sean consistentes.

La memoria caché puede emplear dos procedimientos de escritura:

- **Escritura directa.-** La escritura directa (*Write-Through*) es un esquema que garantiza la consistencia de los datos e instrucciones al actualizar a ambos, la memoria caché y al siguiente nivel inferior de memoria.

- **Escritura de respaldo.-** La escritura de respaldo (*Write-Back*) es un esquema que actualiza a los datos e instrucciones en el bloque de memoria caché, pero no necesariamente o inmediatamente en el siguiente nivel inferior de memoria.

  Cuando existe una escritura en un bloque de memoria caché, el siguiente nivel inferior de memoria no se actualiza y el bloque de memoria caché se marca como un *bloque sucio de memoria caché* [42].

  En algún momento, si se requiere escribir sobre ese mismo bloque de memoria caché, la información contenida en ese bloque sucio de memoria caché se traslada al siguiente nivel inferior de memoria y se garantiza la consistencia de los datos o instrucciones. En ese momento, el bloque de memoria caché deja de ser un bloque sucio [30].

2.3.3. Multi-Procesamiento simétrico

Describe un ambiente multi-core en el cual los cores son idénticos, ejecutan el mismo conjunto de instrucciones, trabajan en ambiente compartido y utilizan el mismo sistema operativo [9, 23]. Las tareas se asignan a uno de los cores sin afectar al desempeño en términos de latencia [17].

En un esquema de multi-procesamiento simétrico, los recursos se asignan a las aplicaciones en lugar de a los cores. Después de que los cores han iniciado, los calendarizadores los perciben de manera equivalente y los threads pueden ejecutarse en cualquier core de manera concurrente y/o paralela de tal manera que el cómputo es disponible a las aplicaciones en todo momento [23]. Para realizar la comunicación inter-core, basta con utilizar primitivas estándar del sistema operativo.

2.3.4. Multi-Procesamiento asimétrico

Describe un ambiente multi-core en el cual los cores son distintos, implementan diferentes conjuntos de instrucciones y corren de manera independiente [9]. En el procesamiento asimétrico, existe una relación maestro-esclavo, en la cual el maestro ejecuta el sistema operativo y los otros cores ejecutan las aplicaciones. En este procesamiento, asignar una tarea puede afectar la latencia debido a la optimización en el desempeño en la distribución de la tarea. Esta optimización se produce por el aumento en la complejidad de la programación pues el programador tiene que lidiar con el espacio (asignación de tareas en múltiples cores) y el tiempo (programación de tareas) [17].

El procesamiento en los cores puede ser [23]:

- **Homogéneo.-** Cada core ejecuta el mismo tipo y versión del sistema operativo.
- **Heterogéneo.-** Cada core ejecuta, o un sistema operativo diferente, o una versión diferente del mismo sistema operativo.

  Para que los sistemas operativos realicen la comunicación inter-core, deben implementar un esquema de comunicación o elegir una infraestructura en común, como la comunicación sobre IP, y para evitar conflictos entre los recursos compartidos, deben proveer mecanismos de acceso.

En un esquema de multi-procesamiento asimétrico, un proceso siempre se ejecuta en un mismo core aunque otros cores se encuentren ociosos. De esta manera, el sistema puede subutilizar ciertos cores, mientras puede sobrecargar a otros. Sin embargo, las aplicaciones se pueden migrar entre los cores (aunque puede ser difícil si los cores ejecutan diferentes sistemas operativos). Este tipo de procesamiento tiene una escalabilidad limitada cuando existen más de dos cores.

2.3.5. Multi-Procesamiento limitado

El multi-procesamiento limitado preserva la abstracción del hardware y la administración del multi-procesamiento simétrico, y provee el control del multi-procesamiento asimétrico en un mismo sistema operativo. Los recursos del sistema y los threads se asignan y comparten en un número delimitado de cores [23].

En un mismo sistema, el multi-procesamiento limitado puede existir con el multi-procesamiento simétrico; de esta manera se puede realizar la migración de threads entre todos los cores, restringir la migración de threads entre ciertos cores y no realizar migración de threads.

Este procesamiento simplifica la migración de código y el diseño de pruebas [31].

Ventajas en comparación a un multi-procesamiento simétrico [23]:

- Las aplicaciones se pueden ejecutar en un número delimitado de cores.
- Las aplicaciones desarrolladas para correr en un core, pueden coexistir con las aplicaciones desarrolladas que corren en múltiples cores.

2.4. Medidas de desempeño

El desempeño en un sistema paralelo se refiere a su capacidad de respuesta - esto es, el tiempo que se requiere para responder a un estímulo (evento) o al número de eventos procesados en un intervalo específico [27].

El estudio del desempeño de programas que se ejecutan en paralelo es importante para evaluar la plataforma de hardware, determinar un mejor algoritmo y examinar los beneficios del paralelismo [11].

El objetivo principal de un sistema paralelo es reducir el tiempo de ejecución de un programa. El tiempo de ejecución depende de diversos factores; entre estos se encuentran, la arquitectura del sistema, el compilador, el sistema operativo, el ambiente de programación, el modelo de programación, las dependencias entre los cálculos y las propiedades del programa, como las localidades de las referencias a memoria. Cuando se desarrolle un programa que se ejecute en paralelo, estos factores se deben considerar. Sin embargo, es difícil considerar a todos los factores [32].

Para facilitar el desarrollo y análisis de los programas que se ejecutan en paralelo, se analizan *medidas de desempeño* que influyan en factores importantes. Estas medidas pueden ser modelos teóricos y/o las mediciones de los tiempos de ejecución para un sistema paralelo específico.

Las medidas que a continuación se describen, tienen por objetivo obtener mediciones para arquitecturas homogéneas.

2.4.1. Tiempo de ejecución

El *tiempo de ejecución serial* de un programa se define como el tiempo transcurrido entre el inicio y el fin de una ejecución sobre un elemento de procesamiento y el *tiempo de ejecución paralelo* se define como el tiempo que transcurre desde el momento en que inicia el cálculo paralelo hasta que el último elemento de procesamiento termine su ejecución [11, 39].

El tiempo de ejecución se considera la medida principal de desempeño que un programa paralelo es capaz de conseguir [26]. De esta manera este trabajo de tesis considera al tiempo de ejecución como la medida de desempeño a utilizar y se calcula como:

$$\textit{Tiempo de ejecución} = \mu$$

Donde, μ es el promedio de un conjunto de ejecuciones realizadas.

2.4.2. Aceleración

Cuando se evalúa un sistema paralelo, *la aceleración* es una medida que indica la ganancia en desempeño de la paralelización de una aplicación en comparación a una implementación secuencial [11].

El procesamiento paralelo debe reducir el tiempo de ejecución de un programa, y por tanto, debe incrementar la aceleración y mejorar el tiempo de ejecución. La aceleración se define como la relación entre el tiempo de ejecución del mejor algoritmo secuencial y el tiempo de ejecución con n elementos de procesamiento [11, 39], y se calcula como:

$$Aceleración = \frac{T_1}{T_n}$$

donde, $T1$ es el tiempo en que un elemento de procesamiento ejecuta una tarea y T_n es el tiempo en que $_n$ elementos de procesamiento ejecutan la misma tarea.

2.4.3. Eficiencia

La eficiencia se define como la relación de la aceleración con el número de elementos de procesamiento que intervienen en la solución paralela y se interpreta como el porcentaje de uso de los N elementos de procesamiento [39]. Se calcula de la siguiente manera:

$$Eficiencia = \frac{Aceleración}{N}$$

donde, N es el número de elementos de procesamiento que intervienen en la solución paralela.

2.4.4. Costo

El *costo* o *trabajo*, se refiere al costo de resolver un problema sobre un sistema paralelo y se define como el producto del tiempo de ejecución paralelo por el número de elementos de procesamiento que intervienen en la solución. El costo refleja la suma del tiempo que cada uno de los elementos de procesamiento gasta al resolver el problema. Se calcula de la siguiente manera [11]:

$$Costo = N \times Tn$$

donde, Tn es el tiempo de ejecución paralelo y N es el número de elementos de procesamiento que intervienen en la solución paralela.

2.5. Resumen

Este capítulo presenta un estudio de la tecnología multi-core. El objetivo al finalizar este capítulo es comprender la estructura y el funcionamiento de la arquitectura multi-core; se detallan las ventajas de la arquitectura en comparación a las arquitecturas single-core, se presentan sus desventajas tanto en los programas, la estructura física y en la temperatura; se detallan y mencionan las aplicaciones donde esta tecnología se puede utilizar y los sistemas en donde se puede implementar, además se indica la manera de explotar el paralelismo a nivel de tareas en una arquitectura multi-core.

Se explica la estructura física y la comunicación de las arquitecturas multi-core, se menciona la estructura jerárquica de la memoria y se analiza la estructura y características del nivel de memoria caché. Se menciona el procesamiento de un sistema operativo multi-tareas y se detalla la explicación del flujo de control en el procesamiento multi-core; así mismo, se describe y detalla el procesamiento en la memoria caché y se explican los tipos de procesamiento que existen en la ejecución paralela de un programa.

Por otro lado, el objetivo también es presentar algunas medidas de desempeño a tomar en cuenta en la evaluación de los programas que se ejecutan en paralelo y se hace énfasis en el tiempo de ejecución como la medida de desempeño a utilizar en este trabajo de tesis.

Capítulo 3

Trabajo relacionado

Este capítulo presenta una revisión del trabajo relacionado en el desempeño en la migración de threads.

Se analizan dos trabajos relevantes. El primero se denomina "Entendimiento del Costo de la Migración de Threads para Aplicaciones Java Multi-Threaded que se ejecutan sobre una plataforma Multi-Core"(*Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform*) y el segundo se denomina "Implicaciones en el Desempeño en la Migración de un Thread sobre un Chip Multi-Core"(*Performance Implications of Single Thread Migration on a Chip Multi-Core*).

El trabajo relacionado se organiza de la siguiente manera: se presenta una introducción al trabajo relacionado; se enfatiza en el problema, los objetivos y en la importancia de la investigación. Posteriormente se describe el planteamiento del escenario de experimentación y se indica el desarrollo y parámetros a evaluar. Luego se presentan los resultados y con ello las conclusiones. Al finalizar, se mencionan las contribuciones del trabajo en estudio.

3.1. Costo de la migración de threads

El trabajo que se denomina "Entendimiento del Costo de la Migración de Threads para Aplicaciones Java Multi-Threaded que se ejecutan sobre una plataforma Multi-Core"(*Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform*) [37] analiza el costo en la migración de threads de aplicaciones en Java para una plataforma multi-core.

El trabajo muestra un análisis del origen de la migración de threads y se estudian los siguientes factores [37]:

- **El comportamiento de la aplicación.-** El tamaño del trabajo en conjunto (*working set size*).
- **El comportamiento del sistema operativo.-** La frecuencia de migración.
- **Las características del hardware.-** La memoria caché no uniforme que se comparte entre los cores (*nonuniform cache sharing among cores*).

Además, se presentan resultados del desempeño que se adquieren de aplicaciones en Java.

3.1.1. Objetivo

Los sistemas multi-core incrementan la dificultad en el análisis del desempeño por la inclusión de un nuevo factor: la migración de threads. El análisis de este factor es importante para entender el costo adicional en el desempeño del paralelismo en sistemas multi-core. El objetivo es averiguar si este factor requiere de cierta atención en el análisis del desempeño en sistemas multi-core.

3.1.2. Experimentación

Arquitectura

Los experimentos se realizan en un servidor HS21 BladeCenter que se constituye de dos procesadores Intel Xeon 5345, cada uno de 4 cores. Cada procesador está organizado con dos chips, cada uno se constituye de dos cores que comparten una memoria caché unificada de 4 MB. La medida de las memorias caché de datos e instrucciones son de 32 KB y la memoria caché compartida de nivel 2 contiene una política de ubicación asociativa de conjunto de 16-bloques (*16-way set associative*), donde la medida de la línea es de 64 bytes (Figura 3.1).

En la experimentación se utilizó Linux 2.6.16.46-0.12-bigsmp.

Desarrollo

Este trabajo investiga:

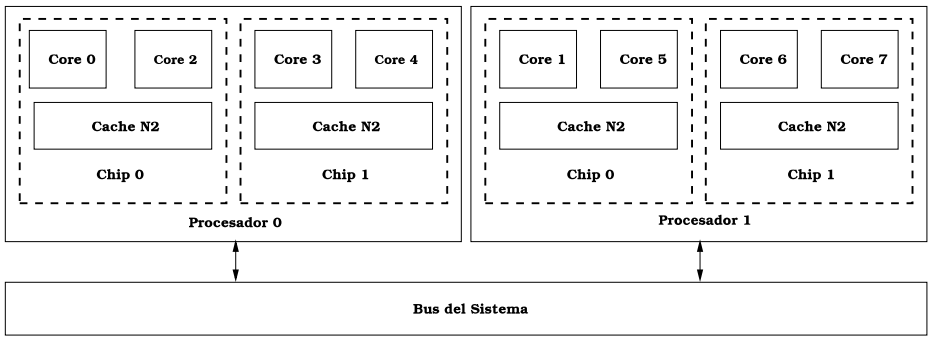

Figura 3.1: Arquitectura multi-core

- Los orígenes del overhead de migración.
- El significado de ese overhead de migración para aplicaciones reales de Java.

La manera en que se estudia el overhead en el desempeño de las migraciones de los threads es por medio del seguimiento de la trayectoria de la ruta de migración de los threads de Java (por ejemplo, entre los dos cores donde ocurre la migración) y los ciclos del procesador consumidos por cada thread.

La información en tiempo de ejecución del kernel y del hardware del procesador multi-core se recolecta con una herramienta desarrollada en Java. Esta herramienta puede configurar los siguientes parámetros:

- **Número de migraciones.-** Es el número de migraciones que ocurren.
- **Ruta de migración.-** Especifica los cores donde ocurre una migración de thread.
- **Intervalo de migración.-** El intervalo de tiempo entre las migraciones.
- **Tamaño del trabajo en conjunto.-** El promedio de datos que residen en memoria caché y son reutilizados entre migraciones.

El overhead en el desempeño en tiempo de ejecución que se introduce por la recolección de los datos es menor al 2 %.

3.1.3. Resultados

Los resultados principales son:

- Los experimentos revelan que el número de migraciones no son un indicador suficiente del overhead de migración.
- Si la frecuencia de migración es baja, el overhead de migración es mínimo independientemente de otros factores.
- El análisis del desempeño se debe enfocar en las migraciones que crucen los límites de los dominios de la memoria caché de nivel 2.
- Los threads cuyo tamaño de trabajo en conjunto es pequeño o es grande, sufren el menor costo en la migración.
- El mayor número de desaciertos en memoria caché es cuando en la migración del thread se tiene un tamaño de trabajo en conjunto que se acerca a ocupar el total de memoria caché.

3.1.4. Conclusiones

Las principales conclusiones son:

- El overhead de migración no se puede explicar con uno o dos factores de desempeño. Los tres factores: frecuencia de migración, las migraciones que cruzan los límites de los dominios de la memoria caché de nivel 2 y el tamaño del trabajo en conjunto contribuyen al overhead de migración.
- El análisis muestra que las aplicaciones en Java estudiadas en ambientes controlados si muestran penalidades en el desempeño, pero en ambientes reales no se sufre de pérdidas relevantes.

3.1.5. Contribuciones

- Un análisis detallado de los origenes del overhead de migración.
- Un conjunto de observaciones que ayudan a clasificar las migraciones de los threads dentro de clases con diferentes implicaciones en el desempeño.
- Una caracterización y análisis del comportamiento de la migración de un conjunto de aplicaciones Java multi-threaded que revelan que el overhead de migración en ambientes operativos reales es pequeño.

3.2. Implicaciones en el desempeño en la migración de un thread

El trabajo que se denomina "Implicaciones en el Desempeño en la Migración de un Thread sobre un Chip Multi-Core"(*Performance Implications of Single Thread Migration on a Chip Multi-Core*) [6] estudia las implicaciones en el desempeño de la migración de un thread sobre una arquitectura multi-core. El trabajo investiga las implicaciones en el desempeño en la migración de actividad.

La migración de actividad es un método que se enfoca en el diseño de potencia. Por ejemplo, la migración de actividad en una arquitectura multi-core puede mejorar la eficiencia de la energía si se transfiere la ejecución de un thread hacia otro core cuya densidad de energía (temperatura) sea menor. El objetivo es distribuir el consumo de potencia en el chip [33].

La investigación aborda los siguiente factores: latencia, frecuencia de migración, subconjunto de recursos que son calentados (*warmed-up*), número de cores y organización jerárquica de la memoria caché.

Se va a entender a un recurso calentado como cualquier componente físico que este realizando un procesamiento, y se va a entender a un recurso en frío como cualquier componente físico que se encuentre apagado (shutdown).

3.2.1. Objetivo

Estudios recientes sugieren que consideraciones de desempeño, energía y temperatura de próximas arquitecturas multi-core pueden necesitar de la migración de actividad.

El objetivo de este estudio es investigar el impacto en el desempeño de la migración de threads para diseñar arquitecturas multi-core con eficientes estrategias de desempeño, energía y temperatura que incluyan la migración de actividad.

3.2.2. Experimentación

Arquitectura

Los experimentos se realizan en las siguientes dos arquitecturas multi-core:

1. Cada core se constituye de dos memorias caché privadas de nivel 1, una de datos y una de instrucciones y una memoria caché compartida

entre dos cores de nivel 2. Ambos niveles son de escritura de respaldo y escritura de asignación (*write-allocate*) (Figura 3.2.a).

Una escritura de asignación es cuando un bloque se escribe en memoria caché [41].

2. Cada core se constituye de dos memorias caché privadas de nivel 1, una de datos y una de instrucciones, una memoria caché privada en cada core de nivel 2 y una memoria caché compartida entre los dos cores de nivel 3. La memoria caché de nivel 1 es una escritura directa sin escritura de asignación (*non write-allocate*) y las memorias caché de nivel 1 y 2 son escrituras de respaldo y de asignación (Figura 3.2.b).

   Una escritura directa sin escritura de asignación es cuando un bloque se escribe en memoria principal sin ser escrito a memoria caché [41].

Las arquitecturas no contienen inclusión de memoria caché. Esto significa que las transacciones de la coherencia de la memoria caché para memorias caché privadas de nivel 2 necesitan verificar las etiquetas de la memoria caché de nivel 1 [3].

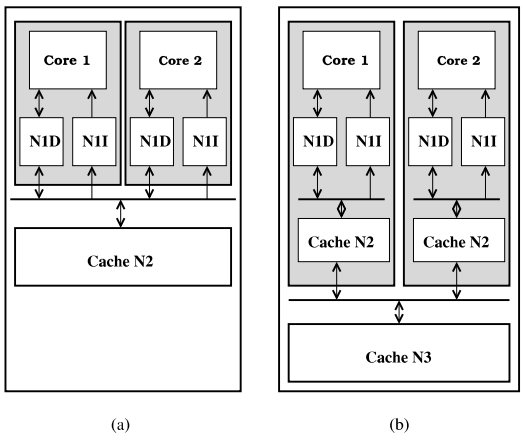

Figura 3.2: Arquitecturas multi-core

Desarrollo

El trabajo investiga las implicaciones en el desempeño en la migración de actividad.

La manera en que se presenta el estudio de la migración es por medio de la simulación de parámetros de diferentes puntos de referencia (*benchmarks*). La experimentación comprende la comparación de migraciones perfectas (donde el overhead de migración es nulo) con diferentes escenarios de migración donde se varía el número de cores en las dos arquitecturas multi-core. Los efectos del periodo de migración se examinan con periodos de migración fijos y aleatorios.

Los valores por omisión en la realización de los experimentos son los siguientes:

- La transferencia de registros se ejecuta en paralelo con la fase de escritura directa.
- La latencia para transferir los registros durante la fase de actualización necesaria es de 100 ciclos; 30 ciclos para la inicialización de la transferencia y 1 ciclo para transmitir cada uno de los 70 registros de la arquitectura.
- La comunicación inter-procesos se calcula con la configuración de hardware en un escenario de migración perfecta.
- El número de cores es 2.

3.2.3. Resultados

Los resultados de la simulación se presentan en dos partes. Primero se reportan los resultados para la arquitectura donde cada core se constituye de dos memorias caché privadas de nivel 1, una de datos y una de instrucciones y una memoria caché compartida entre dos cores de nivel 2 (Figura 3.2.a); después se presentan los resultados para la arquitectura donde cada core se constituye de dos memorias caché privadas de nivel 1, una de datos y una de instrucciones, una memoria caché privada en cada core de nivel 2 y una memoria caché compartida entre los 2 cores de nivel 3 (Figura 3.2.b).

Arquitectura uno

- **Impacto en el desempeño al calentar idealemente un subconjunto de recursos:**
  - Cuando la frecuencia de migración es alta, de 2.5K a 10K ciclos, no deben ser ignoradas las consecuencias de la migración, todos

los recursos necesitan ser calentados para que las consecuencias sean pequeñas.

- Cuando la frecuencia ocurre cada 2.5M ciclos, las pérdidas en el desempeño son pequeñas aún cuando todos los recursos están en frío. Solo un punto de referencia incide en una pequeña pérdida de desempeño cuando el predictor está en frío; entonces, los recursos pueden permanecer en frío (Sección 4.1.2).
- Para frecuencias de migración regulares, entre 40K y 640K ciclos, el comportamiento es impredecible, parece depender de la microarquitectura y de la aplicación.

  Se descubre que un significante overhead se debe a mantener en frío al predictor y además no se obtiene una ganancia considerable al mantener a la memoria caché de nivel 1 caliente. Por eso es importante mantener al predictor de bifurcación caliente, pero no a la memoria caché.

Para el resto de los resultados de esta arquitectura se analiza la frecuencia regular en la migración, pues es donde los puntos de referencia exhiben el comportamiento más impredecible.

- **Impacto en el desempeño al calentar idealemente un subconjunto de recursos para diferentes medidas de tablas:**

  Los resultados de la sección anterior se consideran para una configuración particular. En esta sección se modifica la medida de las memorias caché de nivel 1 y el predictor.

  - A través del estudio de diferentes medidas de memoria caché de instrucciones, se concluye que no es importante mantener una memoria caché de instrucciones caliente. Solo un punto de referencia cada 40K ciclos se comporta con una pérdida de desempeño considerable, cerca del 12 % para una memoria caché de instrucciones de 64KB.
  - La memoria caché de datos caliente no es importante para cualquier medida de memoria caché, frecuencia de migración y punto de referencia.
  - La causa mayor de pérdida del desempeño es mantener al predictor de bifurcación en frío.

Estos resultados muestran que el recurso que se debe mantener caliente es el predictor de bifurcación, y que las memorias caché de instrucciones y de datos tienen consecuencias insignificantes en el desempeño. Solo cuando las latencias de la memoria caché de nivel 2 son de 20 a 28 ciclos, es importante mantener caliente a las memorias caché de nivel 1.

Se concluye que el total del overhead de migración se debe a los efectos en frío y no al overhead de transición, y que el overhead de transición solo tiene significantes contribuciones a la latencia de migración para pequeños periodos de migración menores a 100K ciclos.

Para el resto de los resultados de esta arquitectura, la memoria caché de datos y de instrucciones de nivel 1 se mantienen en frío después de una migración, y la memoria caché de datos de nivel 1 utiliza la política de escritura de respaldo.

- **Componentes del predictor de bifuraciones que son importantes para calentar:**

  En esta sección se estudian componentes de un predictor de bifurcación como las tablas de sentido (*direction tables*), pila de dirección de regreso (*ras*), almacén de objetivo de bifurcación (*BTB*) y memoria caché objetivo (*target cache*).

  - El recurso más importante a calentar es la tabla de sentido de predicción.
  - El siguiente importante recurso a calentar es la memoria caché objetivo y/o el almacén de objetivo de bifurcación.
  - La pila de dirección de regreso es el recurso menos importante a calentar. El impacto en el desempeño es insignificante cuando el recurso está en frío.

  Para el resto de los resultados de esta arquitectura, se mantienen calientes los componentes del predictor, excepto la pila de dirección de regreso.

- **La importancia de dos modos de calentamiento para el predictor de bifurcación: Preparación y Adormecimiento**

  En esta sección se estudian las consecuencias en el desempeño de diferentes modos de calentamiento de un predictor de bifurcación; preparación (*train*), adormecimiento (*drowsy*) y una combinación de ambas en comparación a una migración perfecta.

- En el modo de calentamiento de preparación existe una modesta mejora en el desempeño en comparación al modo en frío; además, con el incremento en la longitud de la fase de preparación existe un incremento en la mejora del desempeño, sin embargo, en comparación a una migración perfecta existen pérdidas en el desempeño.
- En el modo calentamiento de adormecimiento existe una mejora considerable en el desempeño en comparación al modo en frío.
- La combinación de ambos modos de calentamiento resulta en un incremento mínimo al desempeño en comparación a utilizar solo el modo de calentamiento de adormecimiento; pero debido al overhead de energía de la fase de preparación, no parece ser un modo atractivo de calentamiento. Esto significa que en la política de migración, la fase de preparación es innecesaria y por tanto, el bus actualizado de esta fase es innecesario.

Para el resto de los resultados de esta arquitectura, se adopta la política de migración de los resultados citados.

- **El impacto en la migración de threads con el incremento del número de cores:**

  En esta sección se exploran los efectos en el desempeño cuando la migración de threads varia en el número de cores (2, 4 y 8 cores).

  La política de rotación entre los cores que se asume es la política *Mod*. Esta se refiere a rotar cada core $(i+1)\,\%N$; donde i es el core actual y N es el número total de cores sobre la arquitectura multi-core.

  - En general, al incrementar el número de cores, se obtiene un buen desempeño en la migración.

- **Frecuencias de migración aleatoria:**

  Como un intento para generalizar los resultados, se realizan simulaciones donde la frecuencia de migración se determina al usar distribuciones aleatorias.

  - Los datos revelan que entre las corridas con frecuencia de migración aleatoria y las corridas con frecuencia de migración constante existe poca variación, a lo más el 1.3 %. Esto puede indicar que las

corridas con frecuencia de migración constante pueden generalizar los resultados que se obtienen de las frecuencias de migración aleatorias.

- **Bus actualizado:**

  - No es necesario un bus actualizado para preparar al predictor o a las memorias caché.

Arquitectura dos

- **Impacto en el desempeño al calentar un subconjunto de recursos:**

  En esta sección se estudian los efectos en el desempeño cuando se calienta el predictor y/o la memoria caché de nivel 2.

  Para los siguientes resultados, se asume que la migración del thread mantiene a los recursos en frío, solo mantiene al predictor y a las memorias caché de nivel 2 calientes.

  - Si se mantienen los recursos en frío en una migración, se tienen consecuencias serias en el desempeño, a menos que la migración sea infrecuente, cada 2.5M ciclos o más.
  - Para el mejor desempeño es importante mantener caliente al predictor y además a la memoria caché de nivel 2.
  - Pueden existir serias consecuencias en el desempeño si ocurren frecuentes migraciones, cada 10K ciclos o menos.
  - Si las migraciones son infrecuentes, cada 2.5M ciclos o más, se obtiene un desempeño aceptable.
  - Cuando las frecuencias de migración son regulares, entre 40K-640K ciclos, se obtiene un buen desempeño.

  Para el resto de los resultados de esta arquitectura se analiza la frecuencia regular en la migración, pues es donde los puntos de referencia exhiben el comportamiento más impredecible.

- **Influencia con el número de cores:**

  En esta sección se estudia el impacto en el desempeño cuando en la migración del thread se incrementa el número de cores (2, 4 y 8 cores).

Dos enfoques de migración son comparados: mantener a los recursos en frío y mantener al predictor de bifuraciones en modo adormecimiento con la memoria caché privada de nivel 2 coherente (Sección 4.1.2).

  - La estrategia efectiva para tolerar el overhead en la migración de threads con 2, 4 y 8 cores es mantener al predictor de bifuraciones en modo de adormecimiento con la memoria caché privada de nivel 2 coherente.

- **Otros resultados:**

  - Para un mejor desempeño, una arquitectura multi-core que incluye una memoria caché privada de nivel 2 se debe incluir una memoria compartida de nivel 3.
  - Una memoria cache coherente de nivel 2 no puede sustituir a una memoria caché rápida de nivel 3.
  - Los resultados de los experimentos con frecuencias de migración aleatorias, entregan resultados similares a los de la arquitectura 1.

3.2.4. Conclusiones

Los resultados experimentales de los puntos de referencia y de las micro-arquitecturas que son usadas es:

- Las pérdidas en el desempeño por la migración de actividad en una arquitectura multi-core con dos memorias caché privadas de nivel 1, una de datos y una de instrucciones, y una memoria caché compartida entre dos cores de nivel 2 se puede minimizar si:

  - En una migración de un thread la ejecución de un thread continúa sobre un core que anteriormente había sido utilizado.
  - Se recuerda el estado del predictor entre las migraciones.

- Las pérdidas en el desempeño por la migración de actividad en una arquitectura con dos memorias caché privadas de nivel 1, una de datos y una de instrucciones, una memoria caché privada en cada core de nivel 2 y una memoria caché compartida entre los 2 cores de nivel 3 se puede minimizar si:

  - Se recuerda el estado del predictor.

- Se mantienen las etiquetas (*tags*) coherentes de las memorias caché de nivel 2.

  Cada ubicación en memoria caché tiene una etiqueta que contiene el índice del dato en memoria principal que se ha almacenado en memoria caché [7].
- Se permite la transferencia de datos entre memorias caché de nivel 2 desde cores inactivos hacia el core activo.

Los resultados muestran que cuando el periodo de migración es de por lo menos 160K ciclos se tiene un insignificante overhead en la degradación del desempeño por la transferencia del estado del registro entre dos cores y por el vaciado de los datos sucios de la memoria caché de nivel 1.

Los resultados sugieren además, que una estrategia de migración efectiva se puede implementar sin un bus actualizado para la transferencia del estado entre los cores, y sin la fase de preparación para calentar las memorias caché y predictores.

3.2.5. Contribuciones

- Se proporciona a los arquitectos de procesadores multi-core una visión sobre el impacto de la migración de threads en el desempeño.
- Se proporciona la información para diseñar eficientes estrategias térmicas, de energía y de potencia para chips multi-core que soporten la migración de actividad.

3.3. Resumen

Este capítulo presenta una revisión del trabajo relacionado en el desempeño en la migración de threads.

Se analizan dos trabajos relevantes. El primero se denomina Entendimiento del Costo de la Migración de Threads para Aplicaciones Java Multi-Threaded que se ejecutan sobre una plataforma Multi-Core (*Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform*) y el segundo se denomina Implicaciones en el Desempeño en la Migración de un Thread sobre un Chip Multi-Core (*Performance Implications of Single Thread Migration on a Chip Multi-Core*).

El primero investiga el origen del overhead de migración y el significado de ese overhead para aplicaciones reales en Java. Se estudian factores como el tamaño del trabajo en conjunto (*working set size*), la frecuencia de migración

y la memoria caché no uniforme que se comparte entre los cores (*nonuniform cache sharing among cores*).

El segundo estudia las implicaciones en el desempeño de la migración de un thread sobre una arquitectura multi-core y se enfoca en las implicaciones en el desempeño en la migración de actividad. Se estudian factores como latencia, frecuencia de migración, subconjunto de recursos que son calentados (*warmed-up*), número de cores y organización jerárquica de la memoria caché.

El capítulo se estructura de la siguiente manera: se presenta una introducción del trabajo relacionado; se enfatiza en el problema, los objetivos y en la importancia de la investigación con el objetivo de conocer la motivación de estudio y las aportaciones que se pretenden descubrir, posteriormente se describe el escenario de experimentación que comprende las características de la arquitectura y parámetros de configuración, después se indica el desarrollo de la experimentación y se citan los parámetros a evaluar con el propósito de conocer el enfoque de la resolución del problema; en seguida se presentan los resultados y con ello las conclusiones de la investigación, al finalizar se muestra la contribución del trabajo en estudio.

Capítulo 4

Análisis del desempeño por la migración de threads

El objetivo de este capítulo es analizar la migración de threads, describir el escenario de experimentación y especificar la medición del desempeño para realizar la experimentación del trabajo de investigación.

Con el propósito de comprender el proceso y *overhead* en la migración de un thread, se define y detalla el proceso que se requiere para cambiar el flujo de control entre dos cores de procesamiento.

Con la finalidad de detallar el escenario de experimentación y los requisitos necesarios para poder realizar los escenarios de experimentación, con y sin la migración de threads, se especifica la arquitectura, el programa y los parámetros a utilizar en la investigación: la arquitectura multi-core, el microprocesador, el sistema operativo, el kernel, el compilador y las directivas a utilizar, la interfaz de afinidad de thread y el procesamiento multi-core.

Por último y con el objetivo de poder realizar la medición de desempeño, se especifica cómo se presenta la medición del tiempo del programa paralelo y desbalanceado en los dos escenarios de migración.

4.1. Migración de thread

En una arquitectura de memoria compartida se denomina migración de thread al proceso o sucesión de fases que se requieren para cambiar un flujo de control de una unidad de procesamiento a otra.

4.1.1. Fases en la migración de un thread

Una arquitectura multi-core capaz de realizar migración de threads puede conllevar requerimientos adicionales comparados a una arquitectura multi-core que no soporte la migración de threads. En particular cada migración requiere de *correctez* (*correctness*), transferencia de algún estado desde un core hacia otro, tal como registros de la arquitectura, estructuras en la micro-arquitectura como memorias caché y predictores pueden mejorar su desempeño si se preparan; por ejemplo, un recurso sobre un core inactivo se puede preparar por las actualizaciones desde un core en ejecución y/o puede permanecer preparado por la última actualización recibida [6].

Fase de inicialización

Durante la ejecución de un thread sobre un core C1 se presenta una necesidad de migración; por ejemplo, balancear la carga de trabajo en el sistema.

Se decide que la ejecución del thread sobre el core C1 se transfiera a un core inactivo, por ejemplo, core C2. Tan pronto como esta decisión se toma, una señal de migración se envía a los dos cores y C2 se activa. Cuando C2 se activa, comienza *la fase de Inicialización* y los recursos de C2 se encienden. Algunos recursos sobre cores inactivos pueden estar encendidos para conservar sus estados entre activaciones por razones de desempeño, pero si los cores inactivos tienen a todos sus recursos encendidos, se produce un consumo de energía excesivo.

Durante la fase de inicialización en el core C2, el core C1 continua en ejecución normal. Por tanto su desempeño no se perjudica; sin embargo, la longitud de la fase de inicialización pone un límite en la frecuencia de migración. De acuerdo a [20], un core que se encuentra apagado y levante, requiere alrededor de mil ciclos de reloj y, por tanto, cualquier intervalo de migración debe ser más grande que eso [6] (Figura 4.1).

Fase de preparación

Después de la fase de inicialización, se puede necesitar la *fase de preparación*. En esta fase, los recursos de C2, como las memorias caché y predictores, son preparados en base a la ejecución del core C1. Esta fase puede requerir de un bus dedicado y actualizado para la comunicación entre los cores; sin embargo, esta peculiaridad incurre en sobrecarga de energía.

En la cola de la fase de preparación, el pipeline del core C1 se *vacía* (*flushed*) (Figura 4.1).

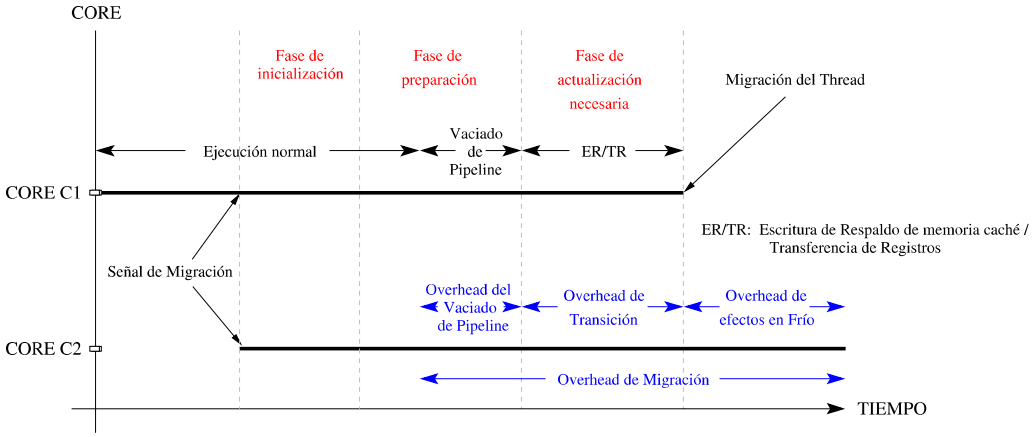

Figura 4.1: Diferentes fases en la migración de un thread desde el core C1 hacia el core C2

Fase de actualización necesaria

En este punto, *la fase de actualización necesaria* puede comenzar. Durante esta fase, el estado del registro de la arquitectura se transfiere desde C1 hacia C2; los bloques sucios de memoria caché en la memoria caché privada de C1 se deben detener y se deben escribir en un nivel de memoria caché inferior. La escritura de respaldo es necesaria para que el core C2 obtenga los datos de la memoria inferior. La longitud de la fase depende del número de registros, latencia y ancho de banda, el número de bloques sucios de memoria caché y la latencia de la escritura de respaldo. Si el registro transferido y el vaciado de la memoria caché ocurren en paralelo, quien dilate más tiempo, determina la latencia final de la fase. Se debe notar, que la latencia de la transferencia de registros es constante, pero la latencia para que se vacíen los bloques sucios de memoria caché no lo es: depende del número de bloques sucios de memoria caché.

Se puede ejecutar primero la escritura de respaldo y después la transferencia de registros. Esta última como una implementación en microcódigo o como una *trampa de software* (*software trap*). Esta rutina debe almacenar los registros en memoria compartida para que el core C2 pueda cargarlos. La transferencia de registros y la escritura de respaldo se pueden realizar también en paralelo, pero se necesita de un bus que comunique los registros entre los cores.

Cuando la fase de actualización necesaria finalice, el core C1 puede que-

dar inactivo y el core C2 puede seguir con la ejecución del thread, o los threads migrados (Figura 4.1).

4.1.2. Otros factores en la migración de un thread

La migración del thread puede degradar el desempeño del sistema debido a que se vacía el pipeline y por la fase de actualización necesaria que intensifica el procesamiento debido a las transiciones de la fase. Por tanto, la latencia de dichas fases debe ser corta.

Después de la migración del thread el desempeño se puede perjudicar debido a los recursos en frío en el nuevo core. Los efectos del arranque en frío son dominados por los desaciertos de arranque en frío de la memoria caché y errores en la predicción de bifurcación. Por eso, la fase de preparación puede ser útil al tener los recursos predispuestos y/o se pueden tener recursos de cores inactivos preservando sus estados entre migraciones.

Se puede notar, que los costos de la migración del thread dependen de la aplicación y de la micro-arquitectura.

Predictor de bifurcación

Una bifurcación es cuando en la ejecución de un programa, el flujo de control se puede alterar de manera condicional. Entonces, un predictor de bifurcación (*Branch Predictor)* es quien predice si la bifurcación se lleva a cabo o no [30].

La certidumbre en el predictor de bifurcación es importante para el desempeño de los microprocesadores. Si el número de etapas del pipeline es grande, la incertidumbre en la predicción de las bifurcación es mayor. Es importante que después de una migración, el predictor tenga una certitud correcta.

Si el predictor se apaga entre migraciones, la fase de preparación puede servir para preparar al predictor. Se puede transmitir la *dirección* (*address*)[1], el *sentido* (*direction*)[2] y la *dirección objetivo* (*target address*)[3] por cada bifurcación que se realiza en el core C1, hacia el core C2.

Entre más tiempo tarde la fase de preparación, se prepara mejor al predictor, pero la fase de preparación debe ser breve, como se menciona anteriormente, incide en costos de energía y en la limitación de la frecuencia de

[1]Es la ubicación de un dato dentro de un arreglo de memoria [30].

[2]Es una decisión binaria que puede tomar los valores referentes a llevar a cabo la bifurcación ó no llevar a cabo la bifurcación [19].

[3]Es la dirección que contendrá el Contador de Programa si una bifurcación se llega a realizar [30].

migración. De tal manera, se necesita un predictor con certitud correcta y cuya preparación sea rápida.

Una alternativa para preparar al core C2 es preservar el estado del predictor entre las migraciones en un estado reducido de adormecimiento de filtración [6, 13].

Memoria caché

Las memorias caché son áreas temporales de retención de datos (ver Sección 2.2.1).

Si después de la migración del thread, el core donde residía el thread cambia su estado a inactivo, el flujo de sus memorias caché privadas se pueden comportar de la siguiente manera [6]:

- **Comportamiento frío.-** Se mantienen con escritura de respaldo, su contenido se vacía a un nivel de memoria compartida inferior y se apagan. Cuando en la memoria caché privada del core activo se obtiene un desacierto, se sondea el bloque faltante en la memoria de nivel jerárquico inferior.

- **Comportamiento coherente.-** Se mantienen activas con escritura de respaldo, mantienen sus etiquetas coherentes y permiten la transferencia de datos entre memorias caché privadas de cores inactivos hacia el core activo. Cuando en la memoria caché privada del core activo se obtiene un desacierto, primero se sondean las memorias caché en los otros cores, si no se encuentra ahí el bloque faltante, se sondea en un bloque de memoria de nivel jerárquico inferior.

4.2. Arquitectura multi-core

Los experimentos se realizan en un servidor PowerEdge T110 que se constituye de un procesador Intel Xeon CPU X3430. Este procesador se compone de 4 cores. Cada core se constituye de dos memorias caché privadas de nivel 1, una de datos de 32-KB y una de instrucciones de 32-KB; también se encuentra una memoria caché de nivel 2 de instrucciones y datos de 256-KB y una memoria caché de 8-MB de nivel 3 compartida entre todos los cores (Figura 4.2) [16].

Los experimentos se realizan en el sistema operativo Linux CentOS versión 5.4 con kernel 2.6.18-164.6.1.el5 y se utiliza el compilador de C++ de

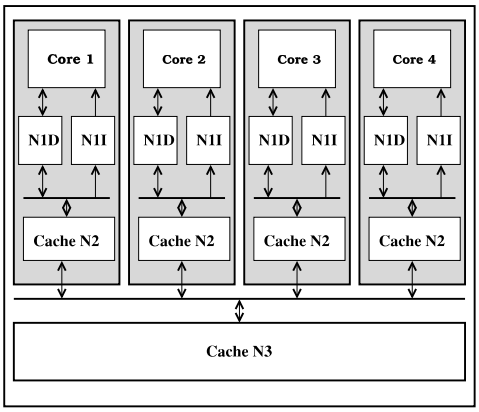

Figura 4.2: Diagrama de la arquitectura multi-core

Intel (icpc) que soporta OpenMP versión 3.0 [1]. El kernel soporta la afinidad de thread y está configurado para funcionar con multi-procesamiento simétrico.

4.3. OpenMP

OpenMP es una interfaz de aplicación de programación (API) que soporta la programación paralela de memoria compartida. Es multi-plataforma y se puede codificar con C/C++ y Fortran [4].

OpenMP soporta el modelo bifurcación-unión (*fork-join*). El programa inicia con la ejecución de un thread, inicial o maestro. Cuando en el transcurso de la ejecución de un programa existe una región paralela, el procesamiento se bifurca en un conjunto de threads que incluye al thread inicial. Al terminar el procesamiento de la región paralela, el conjunto de threads se une al thread inicial (Figura 4.3).

OpenMP se encarga de los detalles de bajo nivel para la creación de threads y para la asignación de trabajo de acuerdo a la estrategia del programador. OpenMP se integra de un conjunto de directivas del compilador, rutinas de la biblioteca en tiempo de ejecución y variables de ambiente. OpenMP proporciona los medios para que el usuario pueda:

- Crear conjuntos de threads para que se ejecuten en paralelo.
- Especificar entre los miembros de un conjunto como compartir el trabajo.

- Declarar variables privadas y compartidas.
- Sincronizar threads, y habilitarlos para realizar operaciones exclusivas (por ejemplo, sin la interrupción de otros threads).

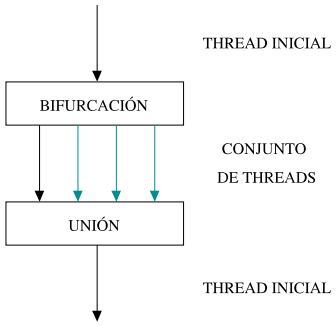

Figura 4.3: Bifuración-Unión. Modelo de programación que soporta OpenMP

4.3.1. Interfaz de afinidad de thread de OpenMP de Intel

La librería en tiempo de ejecución del compilador de OpenMP tiene la capacidad para enlazar threads de OpenMP a unidades de procesamiento físico. Dicha restricción en la ejecución de ciertos threads se hace por medio de la afinidad de threads [38].

La afinidad de threads se soporta en sistemas operativos Windows y las versiones de Linux, y solo los procesadores Intel® soportan la interfaz de afinidad de thread.

Hay tres tipos de interfaces que se pueden especificar para realizar el enlace:

- **Interfaz de afinidad de alto nivel.-** Esta interfaz se controla con la variable de ambiente KMP_AFFINITY. Esta variable determina la topología de la máquina y asigna threads de OpenMP a cores específicos basados sobre la ubicación física en la máquina.
- **Interfaz de afinidad de medio nivel.-** Esta interfaz utiliza una variable de ambiente para especificar en qué cores se pueden ejecutar ciertos threads de OpenMP. Esta interfaz es compatible con la variable de ambiente GOMP_CPU_AFFINITY de GNU gcc y además, se

puede invocar con la variable de ambiente KMP_AFFINITY. Linux solo soporta GOMP_CPU_AFFINITY, pero KMP_AFFINITY, que es soportado por Linux y Windows, realiza funciones similares.

- **Interfaz de afinidad de bajo nivel.-** Esta interfaz usa APIs para habilitar a los threads de OpenMP para hacer llamadas dentro de la librería en tiempo de ejecución para especificar qué cores se utilizan. Esta interfaz es similar a la función *sched_setaffinity* y a funciones sobre sistemas Linux o a la función *SetThreadAffinityMask* y a funciones sobre sistemas Windows. Además, se pueden especificar opciones de la variable de ambiente KMP_AFFINITY que influyen en el comportamiento de la interfaz de la API de bajo nivel.

En este trabajo se utiliza la interfaz de afinidad de bajo nivel para determinar el core en el cual se va a ejecutar cada thread de OpenMP. El archivo de cabecera que contiene la declaración de las funciones de la API de C/C++ de la interfaz de afinidad de bajo nivel es *omp.h*.

Las funciones que se utilizan para la realización del programa se describen a continuación:

1. ```
   void kmp_create_affinity_mask (kmp_affinity_mask_t *mask)
   subroutine kmp_create_affinity_mask(mask)
   integer (kind=kmp_affinity_mask_kind) mask
   ```

   Esta función asigna una máscara de afinidad de thread. Esta función inicializa *mask[4] al conjunto vacío.

2. ```
   int kmp_set_affinity (kmp_affinity_mask_t *mask)
   integer function kmp_set_affinity(mask)
   integer (kind=kmp_affinity_mask_kind) mask
   ```

   Esta función establece la máscara de afinidad al thread en ejecución. El thread se ejecuta en el/los core(s) que se define(n) en *mask.

3. ```
   int kmp_set_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)
   integer function kmp_set_affinity_mask_proc(proc, mask)
   integer proc
   integer (kind=kmp_affinity_mask_kind) mask
   ```

   Esta función añade el core en *mask.

4. ```
   int kmp_unset_affinity_mask_proc (int proc, kmp_affinity_mask_t *mask)
   integer function kmp_unset_affinity_mask_proc(proc, mask)
   integer proc
   integer (kind=kmp_affinity_mask_kind) mask
   ```

   Esta función excluye el core en *mask.

[4]Es el conjunto de identificadores de los cores en el sistema.

5.
```
void kmp_destroy_affinity_mask (kmp_affinity_mask_t *mask)
subroutine kmp_destroy_affinity_mask(mask)
integer (kind=kmp_affinity_mask_kind) mask
```

Esta función retira la máscara de afinidad de thread.

4.4. Programa

Con el objetivo de enfocar el análisis del desempeño por la migración de threads en una arquitectura multi-core, el programa utilizado en el caso de estudio debe cumplir dos propiedades para satisfacer la hipótesis de la investigación: (a) el programa debe ser paralelo y (b) desbalanceado; sin embargo se consideran otras características:

- Se pretende emplear un algoritmo que sea claro, cuyo objetivo sea el análisis del desempeño en la migración de threads y que además sea de fácil comprensión.
- Se emplea un algoritmo que no tenga una relación entre los datos de los threads a migrar. El objetivo es reducir el número de referencias en localidades de memoria donde los threads compartan recursos.
- Se emplea un algoritmo que se aplique en ciencias de la computación. El objetivo es enfocar la investigación al área de estudio.

Con el objetivo de cumplir lo precedente, el programa que se emplea para esta investigación es el producto de una matriz por un vector.

4.4.1. Producto de una matriz por un vector

De forma general, el producto de una matriz $m \times n$ y un vector columna $n \times 1$ se define como el vector columna $m \times 1$ cuyo elemento i-ésimo se obtiene del producto escalar de la i-ésima fila de la matriz por el vector columna $n \times 1$ [10].

Por ejemplo, sea una matriz A de 2×3 y sea un vector X de 3×1.

$$A = \begin{bmatrix} 3 & 2 & -1 \\ 0 & 4 & 3 \end{bmatrix} \qquad y \qquad X = \begin{bmatrix} 2 \\ 1 \\ -3 \end{bmatrix}$$

El resultado del producto de la matriz A por el vector X es un vector de 2×1:

$$AX = \begin{bmatrix} 3 & 2 & -1 \\ 0 & 4 & 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 1 \\ -3 \end{bmatrix} = \begin{bmatrix} 3 \times 2 + 2 \times 1 + (-1) \times (-3) \\ 0 \times 2 + 4 \times 1 + 3 \times (-3) \end{bmatrix} = \begin{bmatrix} 11 \\ -5 \end{bmatrix}$$

4.4.2. Algoritmo paralelo y desbalanceado

El algoritmo del programa paralelo y desbalanceado se explica con el siguiente ejemplo. Sea una matriz A de 6×6 y sea un vector X de 6×1.

$$A = \begin{bmatrix} -3 & -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 & 3 \\ -1 & 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \quad y \quad X = \begin{bmatrix} 2 \\ 1 \\ 0 \\ -1 \\ -2 \\ -3 \end{bmatrix}$$

Algoritmo paralelo

El resultado del producto de la matriz A por el vector X es un vector de 6×1. Para ello cada i-ésimo elemento del vector resultante se obtiene del producto escalar de la i-ésima fila de la matriz A por el vector columna X.

$$AX = \begin{bmatrix} -3 & -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 & 3 \\ -1 & 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \\ -1 \\ -2 \\ -3 \end{bmatrix} = \begin{bmatrix} -16 \\ -19 \\ -22 \\ -25 \\ -28 \\ -31 \end{bmatrix}$$

Cada elemento del vector resultante es independiente entre sí. Por esto, cada elemento del vector resultante se puede obtener de manera simultánea y es así como se paraleliza el programa.

Cada elemento del vector resultante se obtiene de un thread, y cada thread se puede ejecutar en los cores de manera paralela (Figura 4.4).

Algoritmo desbalanceado

Cada elemento del vector resultante se obtiene de un thread, y el conjunto de threads se pueden procesar de manera simultánea y desbalanceada (Figura 4.5).

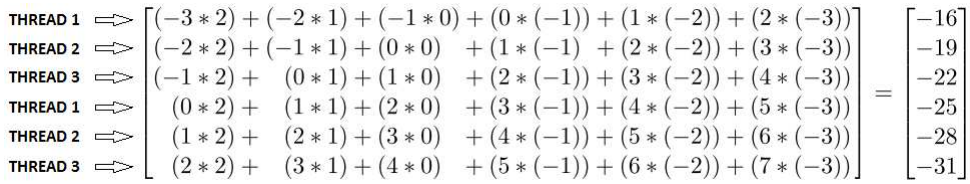

Figura 4.4: Programa paralelo. Procesamiento simultáneo que se realiza por un conjunto de threads para obtener el producto de la matriz por el vector

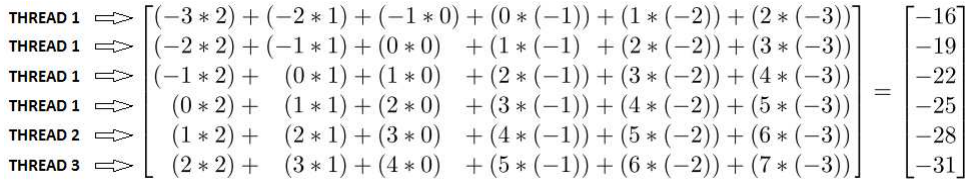

Figura 4.5: Programa desbalanceado. Procesamiento simultáneo y desbalanceado que se realiza por un conjunto de threads para obtener el producto de la matriz por el vector

4.4.3. Código

A continuación se explica el código del programa y se ejemplifica con un caso donde existen 800 threads que se ejecutan en paralelo y un desbalanceo de 400 threads para el core 0, 400 threads para el core 1 y ningún thread para los cores 2 y 3.

El programa se realiza en C++, orientado a objetos, y se estructura en tres clases.

AlgebraLineal.cpp

Este clase contiene la clase principal que representa el punto de inicio de la ejecución del programa y en ella se llama a la función que realiza el producto de la matriz por el vector.

```
#include <iostream>
#include "Matriz.hpp"

using namespace std;

int main()
{

  Matriz m;            // Se asignan los datos de la matriz y del vector.
```

```
  m.MatrizXVector();  // Se realiza el producto de la matriz por el vector.

return 0;
}
```

Matriz.hpp

Este archivo de cabecera declara los identificadores de la clase Matriz.cpp

```
#ifndef _MATRIZ_HPP_
#define _MATRIZ_HPP_

const short int dimension = 800;           // Número de columnas o renglones de la matriz.
const short int numThreads = 800;          // Número de threads totales.
const unsigned long long int dato = 10000; // Valor de cada registro en la matriz y en el vector.

class Matriz
{

private:

  unsigned long long int matriz[dimension][dimension];  // Dimensión de la matriz.
  unsigned long long int vector[dimension];             // Dimensión del vector.
  unsigned long long int sumaTotal[dimension];          // Elemento que contiene al vector resultante.

public:

  void MatrizXVector(void); // Multiplicación matriz por vector.
  Matriz();                 // Asignación de datos.

};

#endif
```

Matriz.cpp

Este clase contiene dos funciones, una función que inicializa los valores de la matriz y del vector, y una función que realiza el procesamiento de multiplicar la matriz por el vector.

En esta clase se realiza el procesamiento de los dos escenarios de migración; para ello, se muestra primero el escenario sin realizar la migración de threads.

Escenario sin migración de threads

```
#include <iostream>
#include <sys/time.h>
#include <omp.h>
#include "Matriz.hpp"

using namespace std;

Matriz::Matriz(void)
{

  // Se almacenan los datos en la matriz.
  for(short int i=0;i<dimension;i++)
  {
    for(short int j=0;j<dimension;j++)
```

```
    {
      matriz[i][j] = dato*dato;
    }
  }

  //Se almacenan los datos en el vector.
  for(short int i=0;i<dimension;i++)
  {
    vector[i] = dato*dato;
  }

}

// Multiplicar matriz por vector.
void Matriz::MatrizXVector(void)
{

  struct timeval tiempoInicial,tiempoFinal;  // Tiempo inicial y final de la medición.
  double tiempo;                             // Tiempo en milisegundos.
  bool bandTiempoInicial = true;

  short int idt;                             // Identificador de thread.
  short int proc;                            // Identificador de core.

  unsigned long long int suma = 0;           // Elemento que contiene la suma de las
                                             // multiplicaciones de la matriz por el vector.
  short int i;
  short int j;

  omp_set_num_threads(numThreads);           // Se establece el número de threads para la región paralela.

  ////////////////////////////////////////
  //
  // Se inicia el procesamiento paralelo.
  //
  ////////////////////////////////////////

  #pragma omp parallel private(idt,proc,i,j)
  {

    kmp_affinity_mask_t mascara;             // Máscara de afinidad de thread.

    kmp_create_affinity_mask(&mascara);      // Se crea la máscara de afinidad de thread.

    idt = omp_get_thread_num();              // Se obtiene el identificador de cada thread.

    if(idt < numThreads/2)
    {
      proc = 0;
    }
    else
    {
      proc = 1;
    }

    // Se asigna un core a la mascara de afinidad de threads.
    if(kmp_set_affinity_mask_proc(proc,&mascara) == 0)
    {
      // Se asigna la máscara de afinidad al thread.
      if(kmp_set_affinity(&mascara) != 0)
      {
        cout << "Fallo: No se asignó la máscara de afinidad al thread" << endl;
      }
    }
    else
    {
      cout << "Fallo: No se asignó el core a la máscara de afinidad" << endl;
    }

    ////////////////////////////////////////
    //
    // Se obtiene el programa desbalanceado.
    //
    ////////////////////////////////////////

    #pragma omp barrier
```

```
    ////////////////////////////////////////
    //
    // Se inicia el procesamiento de medición.
    //
    ////////////////////////////////////////

    // Se inicia la medición de tiempo.
    if(bandTiempoInicial == true)
    {
      gettimeofday(&tiempoInicial, NULL);
      bandTiempoInicial = false;
    }

    // Se recorre por columna.
    #pragma omp for reduction(+:suma)
    for(short i=0;i<dimension;i++)
    {
      // Se recorre por renglón.
      for(short j=0;j<dimension;j++)
      {
        suma += matriz[i][j] * vector[j];  // Se obtiene un elemento del vector resultante
      }

    sumaTotal[i]=suma; // Valores del vector resultante.
    suma = 0;
    }

  kmp_destroy_affinity_mask(&mascara);

  }

  // Se finaliza la medición de tiempo.
  gettimeofday(&tiempoFinal,NULL);
  tiempo = (tiempoFinal.tv_sec - tiempoInicial.tv_sec)*1000 +
  (tiempoFinal.tv_usec - tiempoInicial.tv_usec)/1000.0;
  cout <<  endl << "Se ha tardado: " << tiempo << " milisegundos." << endl << endl;

}
```

En la siguiente clase se codifica el escenario para la migración de threads. Para que esta clase realice la migración, se modifica el código para agregar esa parte y se resalta para su mejor entendimiento.

Escenario con migración de threads

```
#include <iostream>
#include <sys/time.h>
#include <omp.h>
#include "Matriz.hpp"

using namespace std;

Matriz::Matriz(void)
{

  // Se almacenan los datos en la matriz.
  for(short int i=0;i<dimension;i++)
  {
    for(short int j=0;j<dimension;j++)
    {
      matriz[i][j] = dato*dato;
    }
  }

  //Se almacenan los datos en el vector.
  for(short int i=0;i<dimension;i++)
  {
    vector[i] = dato*dato;
```

```
  }

}

// Multiplicar matriz por vector.
void Matriz::MatrizXVector(void)
{

  struct timeval tiempoInicial,tiempoFinal;  // Tiempo inicial y final de la medición.
  double tiempo;                             // Tiempo en milisegundos.
  bool bandTiempoInicial = true;

  short int idt;                             // Identificador de thread.
  short int proc;                            // Identificador de core.

  unsigned long long int suma = 0;           // Elemento que contiene la suma de las
                                             // multiplicaciones de la matriz por el vector.
  short int i;
  short int j;

  omp_set_num_threads(numThreads);           // Se establece el número de threads para la región paralela.

  ////////////////////////////////////////
  //
  // Se inicia el procesamiento paralelo.
  //
  ////////////////////////////////////////

  #pragma omp parallel private(idt,proc,i,j)
  {

    kmp_affinity_mask_t mascara;             // Máscara de afinidad de thread.

    kmp_create_affinity_mask(&mascara);      // Se crea la máscara de afinidad de thread.

    idt = omp_get_thread_num();              // Se obtiene el identificador de cada thread.

    if(idt < numThreads/2)
    {
      proc = 0;
    }
    else
    {
      proc = 1;
    }

    // Se asigna un core a la mascara de afinidad de thread.
    if(kmp_set_affinity_mask_proc(proc,&mascara) == 0)
    {
      // Se asigna la máscara de afinidad al thread.
      if(kmp_set_affinity(&mascara) != 0)
      {
        cout << "Fallo: No se asignó la máscara de afinidad al thread" << endl;
      }
    }
    else
    {
      cout << "Fallo: No se asignó el core a la máscara de afinidad" << endl;
    }

    ////////////////////////////////////////
    //
    // Se obtiene el programa desbalanceado.
    //
    ////////////////////////////////////////

    #pragma omp barrier

    ////////////////////////////////////////
    //
    // Se inicia el procesamiento de medición.
    //
    ////////////////////////////////////////

    // Se inicia la medición de tiempo.
    if(bandTiempoInicial == true)
```

```
{
  gettimeofday(&tiempoInicial, NULL);
  bandTiempoInicial = false;
}

// Se recorre por columna.
#pragma omp for reduction(+:suma)
for(short i=0;i<dimension;i++)
{
  // Se recorre por renglón.
  for(short j=0;j<dimension;j++)
  {
    suma += matriz[i][j] * vector[j];  // Se obtiene un elemento del vector resultante
  }

////////////////////////////////////////
//
// Se inicia la migración del thread.
//
////////////////////////////////////////

  if((idt > 99 && idt < 200) || (idt > 299 && idt < 400))
  {

    proc = 2;

    // Se asigna un core a la mascara de afinidad de thread.
    if(kmp_set_affinity_mask_proc(proc,&mascara) == 0)
    {
      // Se excluye un core a la mascara de afinidad de thread.
      if(kmp_unset_affinity_mask_proc(0,&mascara) == 0)
      {
        // Se asigna la máscara de afinidad al thread.
        if(kmp_set_affinity(&mascara) != 0)
        {
          cout << "Fallo: No se asignó la máscara de afinidad al thread" << endl;
        }
      }
      else
      {
        cout << "Fallo: No se excluyó el core a la máscara de afinidad de thread" << endl;
      }
    }
    else
    {
      cout << "Fallo: No se asignó el core a la máscara de afinidad de thread" << endl;
    }
  }
  else if((idt > 499 && idt < 600) || (idt > 699 && idt < 800))
  {

    proc = 3;

    // Se asigna un core a la mascara de afinidad de thread.
    if(kmp_set_affinity_mask_proc(proc,&mascara) == 0)
    {
      // Se excluye un core a la mascara de afinidad de thread.
      if(kmp_unset_affinity_mask_proc(1,&mascara) == 0)
      {
        // Se asigna la máscara de afinidad al thread.
        if(kmp_set_affinity(&mascara) != 0)
        {
          cout << "Fallo: No se asignó la máscara de afinidad al thread" << endl;
        }
      }
      else
      {
        cout << "Fallo: No se excluyó el core a la máscara de afinidad de thread" << endl;
      }
    }
    else
    {
      cout << "Fallo: No se asignó el core a la máscara de afinidad de thread" << endl;
    }
  }
```

```
      sumaTotal[i]=suma; // Valores del vector resultante.
      suma = 0;
      }

    kmp_destroy_affinity_mask(&mascara);

    }

    // Se finaliza la medición de tiempo.
    gettimeofday(&tiempoFinal,NULL);
    tiempo = (tiempoFinal.tv_sec - tiempoInicial.tv_sec)*1000 +
    (tiempoFinal.tv_usec - tiempoInicial.tv_usec)/1000.0;
    cout <<  endl << "Se ha tardado: " << tiempo << " milisegundos." << endl << endl;

}
```

4.5. Medición del desempeño

Cada ejecución de un programa paralelo en una plataforma determinada provoca un tiempo de ejecución. Sin embargo, si el programa se ejecuta dos o más veces, es poco probable que se produzcan los mismos tiempos de ejecución. Las variaciones en la ejecución de un programa paralelo se deben al no determinismo de su ambiente en ejecución y surgen por compartir los recursos de la computadora entre varios programas [26].

Por lo tanto, para efectos de la presente investigación, el objetivo de las mediciones del desempeño se presentan mediante un tiempo de ejecución promedio representativo a todos los posibles tiempos de ejecución. El tiempo de ejecución promedio se puede utilizar para representar la ejecución del programa paralelo y desbalanceado.

Las mediciones del desempeño que se realizan en este trabajo de investigación son aproximaciones, no son mediciones exactas sin embargo el tiempo de ejecución promedio más el margen de error estimado reflejan un nivel de confianza del 97.5 %; por esto, el valor del promedio es tan cercano al real, como si fuera el obtenido estadísticamente por todas las mediciones reales, y se calcula como [36]:

$$Tiempo\ de\ ejecución = \mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$$

donde, μ es el promedio del conjunto de ejecuciones, $t_{0.975}$ es el coeficiente de confianza, s es la desviación estándar del conjunto de ejecuciones y N es el número de ejecuciones.

El coeficiente de confianza que se aplica en la investigación tomado de tablas para un coeficiente de confianza de 97.5 % es de 2.5.

4.6. Resumen

El objetivo de este capítulo es analizar la migración de threads, describir el escenario de experimentación y especificar la medición del desempeño para poder realizar la experimentación en el siguiente capítulo de la tesis.

Se define y detalla el proceso que se requiere para cambiar el flujo de control entre dos cores de procesamiento con el fin de comprender el proceso y overhead en la migración de un thread.

Por otro lado, se detalla el escenario de experimentación y los requisitos necesarios para poder implementar los escenarios de experimentación. El escenario de experimentación engloba la arquitectura multi-core, el microprocesador, el sistema operativo, el kernel, el programa, el compilador y las directivas a utilizar, la interfaz de afinidad de thread y el procesamiento multi-core.

Se especifica que para este trabajo de investigación, la arquitectura multi-core y los parámetros de configuración son: una arquitectura multi-core Intel de 4 cores, un sistema operativo Linux con kernel 2.6.18-164.6.1.el5, dicho kernel realiza el procesamiento simétrico y soporta la interfaz de afinidad de thread, el programa se realiza en C++, orientado a objetos y con directivas de OpenMP para soportar la programación paralela en memoria compartida, y se usa el compilador icpc de Intel que soporta la interfaz de afinidad de thread. La interfaz de afinidad de thread que se utiliza para enlazar los threads de OpenMP a los cores es la interfaz de afinidad de bajo nivel y se mencionan las funciones que se utilizan.

Además, se describen las dos propiedades que el programa debe satisfacer para cumplir la hipótesis del trabajo de investigación, se ejemplifica el programa, se explica el algoritmo, y se escribe y detalla el código.

Por último y con el objetivo de poder evaluar el desempeño del programa paralelo y desbalanceado en los dos escenarios de migración, se esclarece como se calcula el tiempo de ejecución para este trabajo de investigación.

Capítulo 5

Resultados experimentales

Este capítulo presenta los escenarios de experimentación y los resultados de las mediciones del trabajo de investigación. Con el propósito de obtener la información que influya en la decisión de saber si la migración de threads beneficia al desempeño en la ejecución de un programa paralelo y desbalanceado en la arquitectura multi-core, se plantean cuatro políticas que rigen la realización de los experimentos. Para finalizar se muestran los resultados experimentales.

5.1. Mediciones de las pruebas

Se plantean cuatro políticas que rigen la realización de los experimentos.

5.1.1. Política 1

La política 1 se basa en el peor esquema de desbalanceo que puede existir en la arquitectura para un programa paralelo y desbalanceado. El desbalanceo consiste en iniciar la ejecución de la siguiente manera: en el primer core se tienen todos los threads menos uno, en el segundo core se tiene un thread, y en el tercer y cuarto cores se tienen cero threads.

La migración de threads consiste en balancear el procesamiento de la siguiente manera: El primer core va a distribuir el número de threads entre el segundo, tercero y cuarto cores.

Escenario de experimentación para la política 1

Los escenarios de experimentación para la política 1 se describen por medio de tablas; para ello se describe el significado de cada campo:

- **Número de threads totales.-** Es el número de threads totales que existen en el escenario de experimentación.
- **Identificador de core.-** Es el identificador de cada uno de los cores en la arquitectura multi-core.
- **Programa paralelo y desbalanceado.-** Es el estado inicial de ejecución del programa, desbalanceado y paralelo.
- **Número de threads a migrar.-** Es el número de threads que se migran en la ejecución paralela del programa.
- **Programa balanceado.-** Es el resultado de la migración de threads para obtener en la ejecución paralela del programa el balanceo en el procesamiento.

  A continuación se presentan las siguientes tablas que describen varios casos de la política 1 de desbalanceo hasta el estado balanceado.

Número de threads totales	800			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	799	1	0	0
Número de threads a migrar	599	0	0	0
Programa balanceado	200	200	200	200

Tabla 5.1: Escenario de experimentación para 800 threads en la política 1

Número de threads totales	400			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	399	1	0	0
Número de threads a migrar	299	0	0	0
Programa balanceado	100	100	100	100

Tabla 5.2: Escenario de experimentación para 400 threads en la política 1

Número de threads totales	200			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	199	1	0	0
Número de threads a migrar	149	0	0	0
Programa balanceado	50	50	50	50

Tabla 5.3: Escenario de experimentación para 200 threads en la política 1

Número de threads totales	100			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	99	1	0	0
Número de threads a migrar	74	0	0	0
Programa balanceado	25	25	25	25

Tabla 5.4: Escenario de experimentación para 100 threads en la política 1

5.1.2. Política 2

La política 2 se basa en un esquema donde el número total de threads se distribuye en los dos primeros cores. El desbalanceo consiste en iniciar la ejecución de la siguiente manera: en el primero y segundo core se asigna la misma cantidad de threads, y en el tercero y cuarto core se asignan cero threads.

La migración de threads consiste en balancear el procesamiento de la siguiente manera: El primer core distribuye la mitad de sus threads al tercer core y el segundo core distribuye la mitad de sus threads al cuarto core.

Escenario de experimentación para la política 2

Los escenarios de experimentación para la política 2 se describen por medio de tablas; para ello se describe el significado de cada campo:

- **Número de threads totales.-** Es el número de threads totales que existen en el escenario de experimentación.
- **Identificador de core.-** Es el identificador de cada uno de los cores en la arquitectura multi-core.

- **Programa paralelo y desbalanceado.-** Es el estado inicial de ejecución del programa, desbalanceado y paralelo.
- **Número de threads a migrar.-** Es el número de threads que se migran en la ejecución paralela del programa.
- **Programa balanceado.-** Es el resultado de la migración de threads para obtener en la ejecución paralela del programa el balanceo en el procesamiento.

  A continuación se presentan las siguientes tablas que describen varios casos de la política 2 de desbalanceo hasta el estado balanceado.

Número de threads totales	800			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	400	400	0	0
Número de threads a migrar	200	200	0	0
Programa balanceado	200	200	200	200

Tabla 5.5: Escenario de experimentación para 800 threads en la política 2

Número de threads totales	400			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	200	200	0	0
Número de threads a migrar	100	100	0	0
Programa balanceado	100	100	100	100

Tabla 5.6: Escenario de experimentación para 400 threads en la política 2

Número de threads totales	200			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	100	100	0	0
Número de threads a migrar	50	50	0	0
Programa balanceado	50	50	50	50

Tabla 5.7: Escenario de experimentación para 200 threads en la política 2

Número de threads totales	100			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	50	50	0	0
Número de threads a migrar	25	25	0	0
Programa balanceado	25	25	25	25

Tabla 5.8: Escenario de experimentación para 100 threads en la política 2

5.1.3. Política 3

La política 3 se basa en un esquema donde el número total de threads se distribuye en los tres primeros cores. El desbalanceo consiste en iniciar la ejecución de la siguiente manera: en el primero y segundo core se asigna la misma cantidad de threads, en el tercer core se asigna una cantidad menor de threads que en el primero y segundo core, y en el cuarto core se asignan cero threads.

La migración de threads consiste en balancear el procesamiento de la siguiente manera: El primero y segundo core distribuyen una tercera parte de sus threads al cuarto core.

Como caso especial, en el experimento donde el número total de threads es cien, el primero y segundo core distribuyen diez threads al cuarto core y el tercer core distribuye cinco threads al cuarto core.

Escenario de experimentación para la política 3

Los escenarios de experimentación para la política 3 se describen por medio de tablas; para ello se describe el significado de cada campo:

- **Número de threads totales.-** Es el número de threads totales que existen en el escenario de experimentación.
- **Identificador de core.-** Es el identificador de cada uno de los cores en la arquitectura multi-core.
- **Programa paralelo y desbalanceado.-** Es el estado inicial de ejecución del programa, desbalanceado y paralelo.
- **Número de threads a migrar.-** Es el número de threads que se migran en la ejecución paralela del programa.
- **Programa balanceado.-** Es el resultado de la migración de threads para obtener en la ejecución paralela del programa el balanceo en el procesamiento.

  A continuación se presentan las siguientes tablas que describen varios casos de la política 3 de desbalanceo hasta el estado balanceado.

Número de threads totales	800			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	300	300	200	0
Número de threads a migrar	100	100	0	0
Programa balanceado	200	200	200	200

Tabla 5.9: Escenario de experimentación para 800 threads en la política 3

Número de threads totales	400			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	150	150	100	0
Número de threads a migrar	50	50	0	0
Programa balanceado	100	100	100	100

Tabla 5.10: Escenario de experimentación para 400 threads en la política 3

Número de threads totales	200			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	75	75	50	0
Número de threads a migrar	25	25	0	0
Programa balanceado	50	50	50	50

Tabla 5.11: Escenario de experimentación para 200 threads en la política 3

Número de threads totales	100			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	35	35	30	0
Número de threads a migrar	10	10	5	0
Programa balanceado	25	25	25	25

Tabla 5.12: Escenario de experimentación para 100 threads en la política 3

5.1.4. Política 4

La política 4 se basa en un esquema donde el número total de threads se distribuye en los cuatro cores de manera desbalanceada. El desbalanceo consiste en iniciar la ejecución de la siguiente manera: en el primero y segundo core se asigna la misma cantidad de threads, y en el tercero y cuarto core se asignan la misma cantidad de threads, pero en menor número.

La migración de threads consiste en balancear el procesamiento de la siguiente manera: el primero y segundo core distribuyen la misma cantidad de threads hacia el tercero y cuarto core.

Escenario de experimentación para la política 4

Los escenarios de experimentación para la política 4 se describen por medio de tablas; para ello se describe el significado de cada campo:

- **Número de threads totales.-** Es el número de threads totales que existen en el escenario de experimentación.
- **Identificador de core.-** Es el identificador de cada uno de los cores en la arquitectura multi-core.

- **Programa paralelo y desbalanceado.-** Es el estado inicial de ejecución del programa, desbalanceado y paralelo.
- **Número de threads a migrar.-** Es el número de threads que se migran en la ejecución paralela del programa.
- **Programa balanceado.-** Es el resultado de la migración de threads para obtener en la ejecución paralela del programa el balanceo en el procesamiento.

  A continuación se presentan las siguientes tablas que describen varios casos de la política 4 de desbalanceo hasta el estado balanceado.

Número de threads totales	800			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	300	300	100	100
Número de threads a migrar	100	100	0	0
Programa balanceado	200	200	200	200

Tabla 5.13: Escenario de experimentación para 800 threads en la política 4

Número de threads totales	400			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	150	150	50	50
Número de threads a migrar	50	50	0	0
Programa balanceado	100	100	100	100

Tabla 5.14: Escenario de experimentación para 400 threads en la política 4

Número de threads totales	200			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	75	75	25	25
Número de threads a migrar	25	25	0	0
Programa balanceado	50	50	50	50

Tabla 5.15: Escenario de experimentación para 200 threads en la política 4

Número de threads totales	100			
Identificador de core	Core 1	Core 2	Core 3	Core 4
Programa paralelo y desbalanceado	30	30	20	20
Número de threads a migrar	5	5	0	0
Programa balanceado	25	25	25	25

Tabla 5.16: Escenario de experimentación para 100 threads en la política 4

5.2. Resultados experimentales

En esta sección se muestran los resultados experimentales para las 4 políticas. Las siguientes gráficas muestran una comparación en tiempo entre los dos escenarios de migración: con migración de threads y sin migración de threads. Las tablas presentan los datos de las medidas de tiempo de ejecución promedio en milisegundos que se obtienen de los escenarios de experimentación.

5.2.1. Política 1

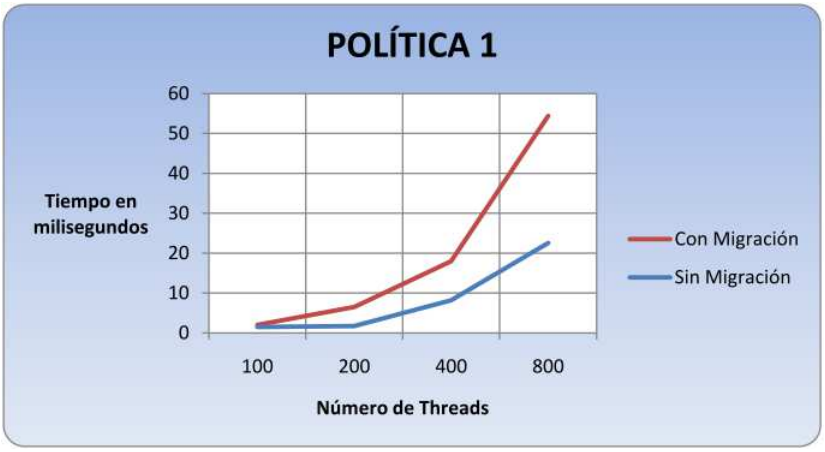

Figura 5.1: Política 1

Número de threads	Con migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$	Sin migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$
100	2.02040 ± 0.12622027	1.50900 ± 0.02902107
200	6.48630 ± 0.82615769	1.74230 ± 0.30378310
400	17.9754 ± 2.69509367	8.18780 ± 0.13004422
800	54.4024 ± 19.4292480	22.5477 ± 1.14106925

Tabla 5.17: Tiempos de ejecución promedio para la política 1

5.2.2. Política 2

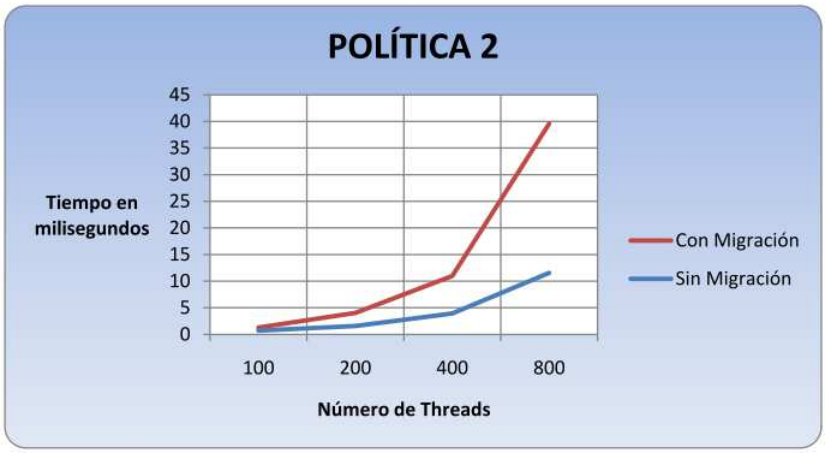

Figura 5.2: Política 2

Número de threads	Con migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$	Sin migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$
100	1.2997 ± 0.11688316	0.7055 ± 0.04587006
200	4.024 ± 0.57133457	1.5732 ± 0.04807951
400	10.9723 ± 1.61901471	3.9303 ± 0.12618387
800	39.5688 ± 4.13838193	11.5524 ± 0.24603585

Tabla 5.18: Tiempos de ejecución promedio para la política 2

5.2.3. Política 3

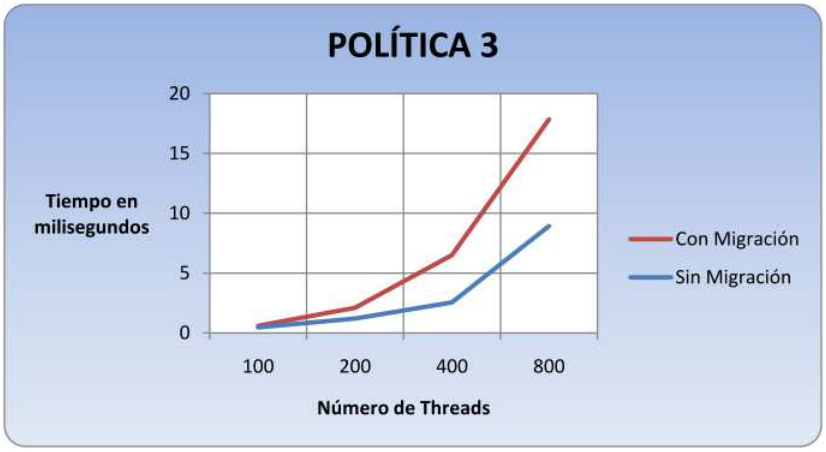

Figura 5.3: Política 3

Número de threads	Con migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$	Sin migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$
100	0.593 ± 0.0479221	0.469 ± 0.03013396
200	2.1067 ± 0.28120229	1.2054 ± 0.06355575
400	6.5031 ± 1.35760003	2.5478 ± 0.05146034
800	17.8388 ± 4.67052085	8.9228 ± 0.63949609

Tabla 5.19: Tiempos de ejecución promedio para la política 3

5.2.4. Política 4

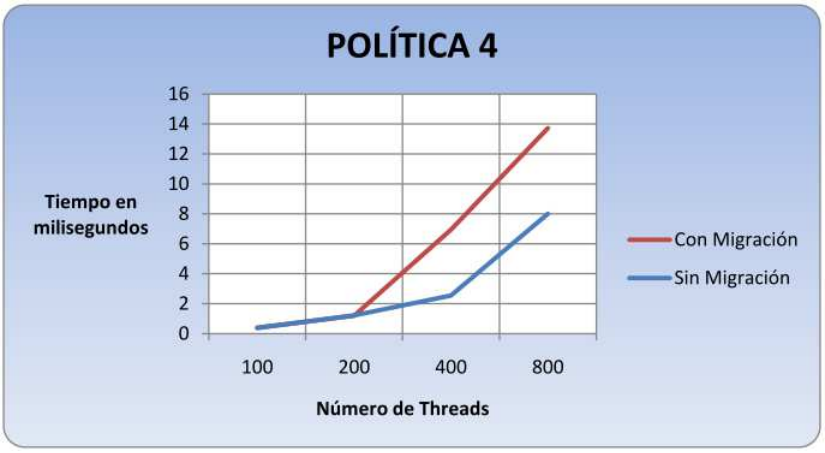

Figura 5.4: Política 4

Número de threads	Con migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$	Sin migración $\mu \pm (t_{0.975})\frac{s}{\sqrt{N-1}}$
100	0.3962 ± 0.01703673	0.371 ± 0.01399405
200	1.1927 ± 0.06122823	1.2199 ± 0.06963042
400	6.9315 ± 1.45287417	2.5406 ± 0.11342288
800	13.7173 ± 2.18194924	7.9979 ± 0.93638485

Tabla 5.20: Tiempos de ejecución promedio para la política 4

5.3. Resumen

Este capítulo presenta los escenarios de experimentación y los resultados de las mediciones del trabajo de investigación.

Con el propósito de obtener la información que influya en la decisión de saber si la migración de threads beneficia al desempeño en la ejecución de un programa paralelo y desbalanceado en la arquitectura multi-core, se plantean cuatro políticas que rigen la realización de los experimentos.

Para finalizar se muestran los resultados experimentales. En este apartado se realizan las pruebas y se presentan las medidas de desempeño por medio de los dos escenarios de ejecución, con migración de threads y sin migración de threads.

Capítulo 6

Conclusiones

Este capítulo expone una evaluación del trabajo de investigación presentado en forma de un resumen crítico, y se retoma la hipótesis y contribución de la tesis para su justificación, se presentan las aportaciones de la investigación y se plantean nuevas vías de estudio.

Con el objetivo de justificar la hipótesis y contribuciones de la tesis, se presenta una recapitulación crítica en la sección de interpretación y análisis de resultados. El análisis de los resultados detalla los resultados experimentales que se obtienen de los escenarios de ejecución.

Con la finalidad de mencionar las aportaciones de este trabajo de investigación, se realiza una comparación con el trabajo relacionado que se cree se acerca más a este trabajo de tesis.

Para finalizar, se plantean nuevos objetivos e identifican nuevas vías de investigación como trabajo a futuro.

6.1. Retomando la hipótesis

Se retoma la hipótesis de la tesis que se presenta en el capítulo 1:

"Dado un programa paralelo y desbalanceado en una arquitectura multicore, se obtiene un mejor desempeño en la ejecución del programa debido a no realizar migración de threads entre los cores"

6.1.1. Discusión

Uno de los objetivos de la migración de threads es mejorar el desempeño del sistema paralelo al balancear la carga de trabajo entre los cores. Dicha migración de threads requiere de un análisis de desempeño que faculte a

los diseñadores de arquitecturas multi-core, plantear algoritmos de balanceo que permitan construir rápidos sistemas multi-core.

Por lo tanto, esta tesis propone un método experimental que se basa en analizar el desempeño de un programa paralelo y desbalanceado en dos escenarios de ejecución, con migración de threads y sin migración de threads. El método se basa en obtener el tiempo de ejecución de cada escenario, y compararlos; el escenario que obtenga el mejor desempeño, es quien entregue los menores tiempos de respuesta. El objetivo del método experimental es conocer si la migración de threads es benéfica al desempeño del sistema multi-core.

El programa que se emplea para esta investigación es el producto de una matriz por un vector. Al principio este programa puede parecer trivial. Sin embargo cumple con las propiedades necesarias para satisfacer la hipótesis de la investigación; pero además se consideran otras características: es de fácil comprensión, se enfoca al área de las ciencias de la computación y es un programa que no tiene una relación entre los datos de los threads a migrar; por lo tanto, estos resultados se pueden extender a programas que presenten independencia entre los datos de sus threads a migrar.

Mientras que los resultados experimentales son limitados a esta micro-arquitectura Intel, los resultados son aplicables para arquitecturas multi-core que sean simétricas a todos los cores en todos los niveles jerárquicos de memoria.

La conclusión que se obtiene para este trabajo de tesis es que cuando el programa paralelo y desbalanceado balancea su procesamiento paralelo mediante la migración de threads, provoca un costo adicional al desempeño, lo que justifica y confirma la hipótesis. Se concluye que se obtiene un mejor desempeño en la ejecución del programa debido a no realizar migración de threads entre los cores.

Las contribuciones que se obtienen de esta investigación son:

- Disminuir la carga a los procesadores al eliminar la migración de threads.
- Disminuir la dificultad de la programación en la comunicación inter-cores al eliminar la migración de threads.
- Disminuir el consumo de energía en los cores al eliminar la migración de threads.

Se espera que los resultados ayuden a los diseñadores de sistemas multi-core para construir mejores sistemas.

6.1.2. Interpretación y análisis de los resultados

La evaluación de los resultados de los ejemplos experimentales del capítulo 5 muestran que, en la mayoría de los casos, un escenario sin migración de threads entre los cores obtiene un mejor desempeño que un escenario con migración de threads entre los cores. La razón principal es que balancear el procesamiento paralelo mediante la migración de threads provoca un costo adicional al desempeño cuando el número de threads totales es mayor y cada vez más desbalanceado. Esto ocasiona que un número excesivo de migraciones de threads se realizen.

Para los escenarios de experimentación de la política 4 se puede concluir que los escenarios sin migración de threads entre los cores es la opción a implementar en una arquitectura multi-core, ya que el beneficio que se obtiene de implementar los escenarios con migración de threads son perjudiciales o insignificantes al desempeño.

Además se concluye que los escenarios con un mayor desbalanceo en comparación a escenarios con un menor desbalanceo impactan más al desempeño. Los escenarios de experimentación muestran que los tiempos de ejecución que se obtienen son mayores cuando el desbalanceo es más desproporcional.

Discusión de resultados

Los resultados experimentales de las políticas 1, 2 y 3 muestran que los escenarios sin migración de threads entre los cores obtienen un mejor desempeño que los escenarios con migración de threads entre los cores.

Los resultados experimentales de la política 4 muestran que para un número mayor de threads, en particular 400 y 800 threads, los escenarios sin migración de threads entre los cores obtienen un mejor desempeño que los escenarios con migración de threads entre los cores. Pero para un número menor de threads, en particular 100 y 200 threads, los tiempos de ejecución no determinan cuál escenario obtiene un mejor desempeño, lo que resulta en poder implementar cualquier escenario de migración.

Para el escenario con 200 threads se obtiene una mejora en promedio del 2.2 % al realizar la migración de threads entre los cores; sin embargo existe un margen de error estimado para el escenario con migración de threads del 5.1 %, y del 5.7 % para el escenario sin migración de threads. Este comportamiento se puede producir porque el escenario de 200 threads se constituye de pocas migraciones y es menor su desbalanceado en comparación a los demás escenarios de migración, lo que provoca no realizar un número excesivo de

migraciones.

En comparación al escenario con 100 threads, se puede deducir que los tiempos de ejecución pueden variar para concluir si los escenarios con migración de threads obtienen un mejor desempeño o viceversa, pero se puede notar que al aumentar el número de migraciones de threads totales, los escenarios sin migración de threads obtienen un mejor desempeño.

El análisis de los resultados experimentales se describen por medio de tablas; para ello se describe el significado de cada campo:

- **Número de threads.-** Es el número de threads que existen en el escenario de experimentación.
- **Escenario de migración.-** Es el escenario de migración que obtiene el mejor desempeño.
- **Tiempo de ejecución.-** Es la diferencia de la medición del tiempo total de los dos escenarios de migración.
- **Deterioro en desempeño.-** Es el porcentaje de deterioro del desempeño al utilizar un escenario de migración.

Número de threads	Escenario de migración	Tiempo de ejecución	Deterioro en desempeño
100	Sin migración	0.51140 milisegundos	33.8 % con migración
200	Sin migración	4.74400 milisegundos	272.2 % con migración
400	Sin migración	9.78760 milisegundos	119.5 % con migración
800	Sin migración	31.8547 milisegundos	141.2 % con migración

Tabla 6.1: Análisis de resultados experimentales de la política 1

Número de threads	Escenario de migración	Tiempo de ejecución	Deterioro en desempeño
100	Sin migración	0.59420 milisegundos	84.2 % con migración
200	Sin migración	2.45080 milisegundos	155.7 % con migración
400	Sin migración	7.04200 milisegundos	179.1 % con migración
800	Sin migración	28.0164 milisegundos	242.5 % con migración

Tabla 6.2: Análisis de resultados experimentales de la política 2

Número de threads	Escenario de migración	Tiempo de ejecución	Deterioro en desempeño
100	Sin migración	0.1240 milisegundos	26.4 % con migración
200	Sin migración	0.9013 milisegundos	74.7 % con migración
400	Sin migración	3.9553 milisegundos	155.2 % con migración
800	Sin migración	8.9160 milisegundos	99.9 % con migración

Tabla 6.3: Análisis de resultados experimentales de la política 3

Número de threads	Escenario de migración	Tiempo de ejecución	Deterioro en desempeño
100	Sin migración	0.0252 milisegundos	6.7 % con migración
200	Con migración	0.0272 milisegundos	2.2 % sin migración
400	Sin migración	4.3909 milisegundos	172.8 % con migración
800	Sin migración	5.7194 milisegundos	71.5 % con migración

Tabla 6.4: Análisis de resultados experimentales de la política 4

Los resultados experimentales muestran que el margen de error estimado por cada política es mayor cuando el tiempo promedio de la ejecución del programa paralelo y desbalanceado es mayor. Sin embargo los escenarios con migración de threads a diferencia de los escenarios sin migración de threads, cuentan con un margen de error estimado que se dispara cuando el número de threads totales es mayor y cada vez más desbalanceado. Esto se debe a que existe un mayor número de migraciones. Dichas migraciones producen en la fase de inicialización, encender y apagar los recursos de manera indeterminada, en la fase de preparación, vaciar el pipeline, en la fase de actualización necesaria, tener un número sucio de bloques de memoria caché desconocidos y una latencia de escritura de respaldo indeterminada, y además se cuenta con el overhead de efectos en frío, lo que provoca una incertidumbre mayor en el tiempo total de la ejecución del programa.

6.2. Comparación con el trabajo relacionado

Estudios previos relacionados al estudio y análisis del desempeño por la migración de threads entre los cores se han realizado; sin embargo en el mejor de mis conocimientos, antes de la realización de este trabajo de

investigación, se desconocía si un mejor desempeño se obtenía al realizar la migración de threads entre los cores al realizar un balanceo de carga por el sistema operativo. Asimismo, este trabajo de investigación se distingue a los anteriores en que se presentan resultados reales en un hardware real, no son resultados de simulaciones o resultados en ambientes controlados.

A continuación se presenta una comparación con el trabajo relacionado que más se acerca a este trabajo de tesis:

- El trabajo denominado "Entendimiento del Costo de la Migración de Threads para Aplicaciones Java Multi-Threaded que se ejecutan sobre una plataforma Multi-Core"[37] se enfoca en calcular el impacto en el desempeño de la migración de threads entre los cores mediante aplicaciones Java multi-threaded. Esta trabajo investiga cuáles son los orígenes del overhead de migración y el significado de este overhead en aplicaciones Java; sin embargo, los resultados se obtienen de ambientes controlados.

  La tesis se distingue de este trabajo relacionado en que no se investigan los orígenes del overhead de migración, ni el significado de ese overhead en aplicaciones Java. Lo que se investiga es el costo de la migración, el cual parte de un desbalanceo inicial.

- El trabajo denominado "Implicaciones en el Desempeño en la Migración de un Thread sobre un Chip Multi-Core"[6] estudia las implicaciones en el desempeño por la migración de un thread al variar diversos parámetros cuando se corren aplicaciones sobre un simulador. El enfoque de la investigación tiene por objetivo estudiar la migración de actividad.

  La tesis se distingue de este trabajo relacionado en que esta investigación analiza el desempeño en la migración de threads, entretanto el trabajo relacionado estudia el desempeño en la migración de un thread; por otro lado los resultados que se obtienen son resultados reales, no son simulaciones. El objetivo del trabajo relacionado se enfoca en la migración de actividad. Mientras tanto el objetivo de esta tesis se enfoca en ponderar el efecto de la migración de threads en el desempeño.

  Cabe destacar que el trabajo relacionado menciona como una vía de trabajo a futuro al análisis del desempeño por la migración de threads en una arquitectura multi-core.

6.3. Trabajo a futuro

Este trabajo tiene como objetivo analizar el desempeño de una arquitectura multi-core al analizar un programa paralelo y desbalanceado que no contiene relación entre los datos a migrar y que se realiza en dos escenarios de ejecución, con migración de threads y sin migración de threads. El estudio del análisis del desempeño se obtiene por las mediciones de los tiempos de ejecución; sin embargo, se plantean nuevos objetivos y se identifican nuevas vías de investigación:

- **Análisis del desempeño por la migración de threads en escenarios multi-threading.-** El estudio y evaluación del comportamiento del desempeño se puede extender a escenarios hiper-threading donde los programas se ejecutan simulando dos cores lógicos en uno físico. Se pueden considerar variaciones de plataformas multi-core, y comparaciones de escenarios hiper-threading, sin migración de threads y con migración de threads. La intención es conocer si los sistemas multi-core se benefician de la migración de threads en escenarios que utilicen hiper-threading.

- **Análisis del desempeño por la migración de threads en programas que contengan una estrecha relación con los datos a migrar.-** El estudio y evaluación del comportamiento del desempeño se puede extender a escenarios que contengan programas con una estrecha relación entre los datos a migrar. Se pueden considerar variaciones de plataformas multi-core, y comparaciones de escenarios con hiper-threading, sin migración de threads y con migración de threads. La intención es incluir referencias a localidades de memoria donde los threads compartan recursos; con ello se pretende saber si los sistemas multi-core se benefician de la migración de threads al ejecutar programas con una estrecha relación entre los datos.

- **Análisis del desempeño por la migración de threads en multi-procesadores.-** El estudio y evaluación del comportamiento del desempeño se puede extender a multi-procesadores constituidos de procesadores multi-core. Se pueden considerar variaciones de plataformas de multi-procesadores simétricos o de multi-procesadores asimétricos, y se pueden realizar comparaciones de escenarios hiper-threading, sin migración de threads y con migración de threads. La intención es conocer si los multi-procesadores se benefician de la migración de threads.

- **Análisis del desempeño por la migración de threads en memoria distribuida.-** El estudio y evaluación del comportamiento del desempeño se puede extender a procesadores multi-core de memoria distribuida, multi-procesadores simétricos de memoria distribuida, multi-procesadores asimétricos de memoria distribuida o en general sistemas de memoria distribuida, y se pueden realizar comparaciones de escenarios hiper-threading, sin migración de threads y con migración de threads. La intención es conocer si los sistemas de memoria distribuida se benefician de la migración de threads.

Bibliografía

[1] Advanced Compilers and Libraries. Intel® Composer XE 2011 For Windows* and Linux* http://software.intel.com/sites/products/collateral/XE/composer_xe_brief.pdf 2010.

[2] Barbic, J., Multi-core architectures, 2007.

[3] Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R. y Verghese, B. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000.

[4] Chapman, B., Jost, G., Der, R. y Shaheed, M., Using OpenMP. Portable Shared Memory Parallel Programming 2008.

[5] All About Cores. What You Need to Know. Computers & Electronics. http://www.smartcomputing.com/editorial/article.asp?article=articles%2F2008%2Fs1907%2F09s07%2F09s07.asp&articleid=48417&guid= 2008.

[6] Constantinou, T., Sazeides, Y., Michaud, P., Fetis. D. y Seznec, A., Performance Implications of Single Thread Migration on a Chip Multi-Core. Department of Computer Science. University of Cyprus, Nicosia, Cyprus Campus de Beaulieu 35042, Rennes Cedex, France, 2005.

[7] CPU Cache. http://en.wikipedia.org/wiki/CPU_cache 2011.

[8] Dale, N. y Lewis, J., Computer Science Illuminated. Jones and Bartlett Publishers Canada, 2011.

[9] Embedded Multicore: An Introduction. Freescale Semiconductor, 2009.

[10] García, F., Lecciones Prácticas de CALCULO NUMERICO. Universidad Pontificia Comillas. 1995.

[11] Grama, A., Gupta, A., Karypis, G. y Kumar, V., Introduction to Parallel Computing. Addison Wesley, 2003.

[12] Hartley, S., Concurrent Programming The Java Programming Language, Oxford University Press, 1998.

[13] Heo, S., Barr, K. y Asanovic, K., Reducing power density through activity migration. En ISLPED '03: Proceedings of the 2003 International Symposium on Low Power Electronics and Design, 2003.

[14] Hitchcock, R., Multi-core CPUs. http://www.windowsnetworking.com/articles_tutorials/Multi-Core-CPUs.html 2007.

[15] Intel® Multi-Core Processor Architecture Development Backgrounder. Intel, 2005.

[16] Intel® Xeon® Processor 3400 Series. Datasheet - Volume 1. Intel http://www.intel.com/Assets/PDF/datasheet/322371.pdf 2010

[17] Karam, L., AlKamal, I., Gatherer, A., Frantz, G., Anderson, D. y Evans, B., Trends in Multi-Core DSP Platforms. Electrical Engineering Dept., Arizona State University, Texas Instruments, School of Electrical and Computer Engineering, Georgia Tech, Atlanta y Electrical & Computer Engineering Dept., The University of Texas at Austin, 2009.

[18] Kazempour, V., Fedorova, A. y Alagheband, P., Performance Implications of Cache Affinity on Multicore Processors, Simon Fraser University, Vancouver Canada, 2008.

[19] Kim, H., Joao, J., Lee, J., Patt, Y. y Cohn, R., Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware, IEEE Transactions on Computers, Vol. 56. No. 9. 2009.

[20] Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P. y Tullsen, D., Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures. Computer Architecture Letters, 2 Abril 2003.

[21] Lonsing, F., Pthreads Programming in C. FMV, SS, 2009.

[22] Multi-Core Technology. http://www.tech-faq.com/multi-core-technology.html 2010.

[23] Multicore Processing. http://www.qnx.org/developers/docs/6.3.2/neutrino/sys_arch/smp.html#AMP 2010.

[24] Multi-core processor. http://en.wikipedia.org/wiki/Multi-core_processor 2010.

[25] Multimedia http://es.wikipedia.org/wiki/Multimedia 2011

[26] Ortega, J., Architectural Patterns for Parallel Programming. Models for Performance Estimation, 2006.

[27] Ortega, J., Patterns for Parallel Software Design. John Wiley & Sons Inc - Estados Unidos, 2010.

[28] Pancake, C. M. y Bergmark, D., Do Parallel Languages Respond to the Needs of Scientific Programmers?. Computer Magazine, IEEE Computer Society, 1990.

[29] Pancake, C. M., Is Parallelism for You? Oregon State University. IEEE Computational Science and Engineering, Vol. 3, No. 2. 1996.

[30] Patterson, D. y Hennessy, J., Computer Organization and Design. THE HARDWARE / SOFTWARE INTERFACE. 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA: Morgan Kaufmann Publishers, 2009.

[31] QNX Software Systems, 2007.

[32] Rauber, T. y Rünger, G., Parallel Programming for Multicore and Cluster Systems. Springer, 2010.

[33] Sazeides, Y., Liqiang, H., Michaud, P. y Seznec, A., High Performance Activity Migration for Thermally Constrained Single Chip Multi-Cores. Department of Computer Science, University of Cyprus, Nicosia, Cyprus y IRISA/INRIA, Rennes, France, 2008.

[34] Schirrmeister, F., Multi-core Processors: Fundamentals, Trends, and Challenges. Imperas, Inc., Geng Road, Palo Alto, California, 2007.

[35] Siddha, S., Pallipadi, V. y Mallik A., Process Schedulling Challenges in the Era of Multi-Core Processors. Intel Technology Journal, pág. 361, 2007.

[36] Spiegel, M., Teoría y problemas de estadística. Serie de compendios Schaum. McGraw-Hill, 1970.

[37] Teng, Q., Sweeney, P. y Duesterwald, E., Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform, IEEE, 2009.

[38] Thread Affinity Interface (Linux* and Windows*). http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/mac/optaps/common/optaps_openmp_thread_affinity.htm 2010.

[39] Tokhi, M., Hossain, M. y Shaheed, M., Parallel Computing for Real-time Signal Processing and Control. Springer, 2003.

[40] TMurgent Technologies. White Paper Processor Affinity Multiple CPU Scheduling. http://www.tmurgent.com/WhitePapers/ProcessorAffinity.pdf 2003.

[41] Write Allocate on Miss. http://www.cs.umass.edu/ weems/CmpSci635A/Lecture10/L10.26.html

[42] Write-through and Write-back cache. http://forums.techarena.in/motherboard-processor-ram/1297414.htm 2005.

[43] Zangler, T., Optimisation: Operating System Scheduling on multi-core architectures. Seminar "Parallel Computinig", 2008.