UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

Estudio del comportamiento de programas multihilo en sistemas NUMA

Autor/a:
Javier Beiro Piñón

Titores:
Oscar García Lorenzo
Ruben Laso Rodríguez

Grao en Enxeñaría Informática

Julio 2023

Trabajo de Fin de Grado presentado en la Escola Técnica Superior de Enxeñaría de la Universidade de Santiago de Compostela para la obtención del Grado en Ingeniería Informática

D. Oscar García Lorenzo, Profesor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, y **D. Ruben Laso Rodríguez**, Investigador del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela,

INFORMAN:

Que la presente memoria, titulada *Estudio del comportamiento de programas multihilo en sistemas NUMA*, presentada por **D. Javier Beiro Piñón** para superar los créditos correspondientes al Trabajo de Fin de Grao da titulación de Grao en Ingeniería Informática, se realizó bajo nuestra tutoría no Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

Y para que así conste a los efectos oportunos, expiden el presente informe en Santiago de Compostela, a 3 de julio de 2023:

Tutor/a, Cotutor/a, Alumno,

Oscar García Lorenzo Ruben Laso Rodríguez Javier Beiro Piñón

Agradecimientos

Quisiera empezar expresando mi más sincera gratitud a aquellos que me han apoyado a lo largo de todo mi viaje académico. Este trabajo de fin de grado es el fruto de la paciencia, dedicación y apoyo constante de muchas personas a lo largo de estos años.

Primero, me gustaría agradecer a mi familia. A mis padres, por su amor y confianza incondicional, por recordarme siempre que podía llegar más lejos si me lo propongo. A mi hermana y a mis tíos, por su constante apoyo y su creencia en mis capacidades.

Mis amigos del grupo P, cada uno de vosotros ocupa un lugar especial en mi corazón y siempre me tendréis como amigo. Quiero agradecer especialmente a Martín y a Nerea, mis primeros amigos en el grado. Martín, te agradezco por convertirte en un compañero de vida, por ser esa persona a la que siempre podré molestar con los conceptos más aburridos y por ser mi apoyo en los momentos más difíciles. Nerea, sin ti no habría sobrevivido a primero, me alegro mucho de haber podido compartir contigo tantas tardes de "más café".

También me gustaría expresar mi agradecimiento al grupo de investigación del CiTIUS. A Fran, Oscar, Caba, Tomás y Juan Carlos, por su orientación y apoyo, por ayudarme siempre que estaba perdido entre *papers* y resultados. Quiero dar un agradecimiento especial a Ruben, por su paciencia y dedicación, por mostrarme cómo convertir la curiosidad en conocimiento y por hacer que el proceso de investigación fuese no solo útil si no también divertido, además de por aguantar todas las dudas que he tenido (que no fueron pocas).

Estoy infinitamente agradecido por haber tenido la oportunidad de aprender y crecer con cada uno de vosotros.

Gracias a todos.

Resumen

Los sistemas NUMA suponen una alternativa muy escalable y de gran interés en el ámbito de los sistemas de memoria compartida. La comprensión de su comportamiento en cuanto a la localidad de los datos y la afinidad de los hilos a estos es crucial para una ejecución eficiente.

Es por ello que, en este trabajo, llevamos a cabo un examen detallado del comportamiento de los sistemas multihilo de memoria compartida NUMA, focalizando nuestro estudio en el impacto que tiene la disposición de dos hilos en los nodos en relación con los datos a los que acceden para el caso de en programas donde predomina la intensidad de las operaciones de acceso a memoria frente a la intensidad computacional. Por un lado, estudiamos la localidad analizando el desempeño de un único hilo en ejecución, realizando operaciones de memoria locales y remotas. Posteriormente, introduciremos un segundo hilo, observando posibles efectos debidos a la compartición de recursos. Finalmente, analizamos la afinidad estudiando los efectos producidos por el acceso a un vector de datos compartidos por dos hilos en paralelo.

Las pruebas han sido realizadas en un sistema NUMA de interconexiones homogéneas, con 4 nodos interconectados en anillo. Los resultados muestran que, en primer lugar, el acceso local a los datos ofrece un rendimiento hasta 3 veces superior a los accesos remotos. En segundo lugar, la compartición de recursos, particularmente de los buses de memoria y el bus de interconexión de nodos, pueden penalizar significativamente el rendimiento cuando los programas paralelos son intensivos en acceso a datos. En tercer lugar, el rendimiento en los accesos a datos compartidos se puede ver influenciado por los protocolos de coherencia, triplicando el tiempo de ejecución de hilos que acceden a sus datos locales al sincronizarse con el hilo que realice el acceso desde un nodo remoto. Por último, los efectos de esta sincronización se pueden mitigar introduciendo un retardo en la ejecución de uno de los hilos, limitando los efectos de conflictos en accesos a memoria y contención de bus. Esto permite que el hilo adelantado se aproxime al rendimiento de una ejecución aislada y que el hilo atrasado aproveche la precarga de datos del primero, mejorando su rendimiento pese al retardo introducido.

De esta manera, este estudio identifica situaciones de pérdida de rendimiento, pudiendo así aplicar posibles correcciones a través de técnicas de planificación de accesos, migración de hilos y/o datos, etc.

Índice general

Índice de figuras

Lista de Tablas

Capítulo 1

Introdución

El constante aumento del número de núcleos y de la densidad de transistores en los microprocesadores actuales ha propiciado la aparición de distintas problemáticas. En concreto, destaca el incremento de la contención en los buses de comunicación y retardos en la propagación de las señales dentro de los propios bancos de memoria caché [3].

Ante esta situación, las arquitecturas NUMA (*Non-Uniform Memory Access*) constituyen una posible solución, permitiendo un acceso a memoria más eficiente y escalable en sistemas multinúcleo [4]. No obstante, este tipo de arquitecturas también presenta distintos retos a la hora de realizar ejecuciones de programas paralelos, donde las distintas distribuciones de hilos y datos sobre los nodos pueden conllevar tiempos de ejecución dispares [5].

En este contexto, resulta de gran interés caracterizar los incrementos o las reducciones de rendimiento que conllevan cada una de estas configuraciones. De esta manera, podría ser posible determinar aquellas que proporcionan una mayor afinidad, permitiendo así obtener ejecuciones más eficientes sin necesidad de variar la algoritmia o lógica de los programas.

1.1. Contextualización

En esta sección previa al contexto de la investigación, se profundizará en los conceptos de sistema **NUMA** y **afinidad**. Indicando la relevancia de la afinidad en estos sistemas.

1.1.1. Sistemas NUMA (*Non Uniform Memory Access*)

Desde hace décadas, el número de transistores dentro de los procesadores ha ido en continuo aumento, contribuyendo a la llegada de procesadores multinúcleo CMP (*Chip Multiprocessor*) y al incremento de la frecuencia de reloj de cada núcleo. A medida que los requerimientos de ancho de banda de los *cores* ha ido

creciendo, este incremento de transistores también se ha utilizado para integrar jerarquías de memoria más profundas y mayores [15, 16].

Sin embargo, este incremento del número de transistores en los microprocesadores también ha propiciado distintas problemáticas asociadas a, por ejemplo, la compartición de recursos entre los distintos cores del *chip* o el retardo de propagación de señales en cachés de gran densidad.

Por un lado, dado que los *cores* no son totalmente independientes sino que pertenecen a un mismo multiprocesador, suelen compartir recursos como el controlador y bus de memoria, o distintos niveles de la jerarquía caché. De esta forma, el tiempo de ejecución de un hilo depende en gran medida de qué otros se encuentran en ejecución en el resto de *cores*. Esto es especialmente relevante en programas intensivos en acceso a memoria, donde, en general, compiten por el espacio disponible en la caché de último nivel (LLC), además de por el controlador y bus para obtener los datos de memoria principal [17, 4].

Por otro lado, esta compactación del tamaño de los transistores también ha provocado problemas como el *wire delay*. El tamaño de las propias conexiones en el interior del chip entre transistores también se ha reducido, lo que resulta en mayores retardos en la propagación de las señales (dado que la resistencia a la transmisión es inversamente proporcional al ancho de las conexiones [3]). Así, el coste de acceso a un banco específico de caché aumenta.

En este contexto en el que el incremento de la densidad de las cachés resulta finito, el aumento del número de núcleos dentro de un mismo procesador puede generar problemas de contención de los recursos y la escalabilidad cada vez es un factor más relevante, surgen las arquitecturas NUMA (*Non-Uniform Memory Access*). Estos sistemas constituyen una posible solución, permitiendo un acceso más eficiente y escalable a memoria en sistemas multinúcleo [4].

Los procesadores multinúcleo modernos, con un controlador de memoria en el *chip*, forman la base para los multiprocesadores NUMA. Estos sistemas están compuestos por nodos interconectados a través de una red dedicada, posibilitando que cada procesador (o nodo) tenga acceso directo a un segmento de la memoria física, mientras que el acceso a las ubicaciones restantes se realiza a través del controlador de memoria de otros procesadores [6]. De esta manera, para los *cores* del computador, el coste de realizar accesos a direcciones locales es menor que si se realizan a direcciones remotas.

Así, al utilizar bancos de memoria diferentes conectados por buses separados, los sistemas NUMA pueden mitigar la contención de los buses cuando se accede a espacios de memoria controlados por nodos distintos. Adicionalmente, estos sistemas no sólo facilitan una distribución eficiente de la memoria principal, sino también de la memoria caché. Esto permite alcanzar capacidades de caché superiores sin necesidad de incrementar su densidad, lo que representa un significativo valor añadido.

1.1.2. El concepto de afinidad en sistemas NUMA

En los sistemas NUMA el concepto de afinidad, se encuentra íntimamente ligado a la asignación óptima de tareas a nodos para lograr la máxima eficiencia. Este concepto está relacionado con la velocidad y el *overhead* de los recursos del nodo requeridos por la tarea [2]. En entornos NUMA y de computación paralela, se puede caracterizar por tres aspectos clave que tienen una influencia directa sobre el rendimiento: los recursos limitados de cada nodo, el estado de la jerarquía de memoria del sistema y la correlación entre distintas tareas en ejecución.

En nuestro caso particular, la afinidad vendrá determinada por los siguientes:

- Los recursos disponibles para cada tarea: cada nodo cuenta con unos recursos limitados disponibles. Cada tarea se ejecutará con un conjunto específico de recursos, que podrán ser o no compartidos con otras tareas en función del nodo de computación en el que se ejecuten.

- El estado de la jerarquía de memoria del sistema: al realizarse accesos no uniformes a memoria, existen diferencias de eficiencia relacionadas con el nodo en el que se encuentra la tarea, dependiendo de en qué caché, o memoria de los distintos nodos, se encuentre los datos, código o información de ejecución.

- Correlación entre distintas tareas con datos compartidos: en gran parte de los casos, el empleo de programas multihilo implica la utilización de memoria compartida, a la que acceden distintos hilos para realizar escrituras y/o lecturas. La modificación de la proximidad de uno de los hilos que acceden a los datos puede influenciar de forma muy significativa al resto.

Por lo tanto, en el desarrollo de este estudio, la afinidad se definirá en base a la disposición de los hilos entre los nodos, la organización de los hilos con respecto a los datos a los que tienen acceso y la configuración relativa de los hilos que operan en paralelo entre sí.

1.1.3. La importancia de la afinidad

Dado que los sistemas NUMA tienden a escalar significativamente en términos del número de hilos, resulta crucial comprender cómo la afinidad de los hilos entre sí, con los nodos y con los datos, afecta al rendimiento del programa.

Específicamente, conocer la distribución más afín de los hilos y los datos en el sistema puede permitir:

- Diseñar algoritmos y estrategias de programación eficientes, aprovechando la arquitectura del sistema y la distribución de la memoria para minimizar los tiempos de acceso y, en consecuencia, maximizar el rendimiento [7].

- Tomar decisiones informadas sobre la asignación de tareas a los nodos de computación, buscando que se ejecuten en aquellos donde puedan desempeñarse de manera más eficiente, teniendo en cuenta las limitaciones de recursos y la jerarquía de memoria [8].
- Evaluar y ajustar en tiempo de ejecución la configuración del sistema para equilibrar la carga de trabajo entre los nodos, evitando cuellos de botella y mejorando la eficiencia general del sistema [9, 10].

Es decir, conocer el rendimiento de los programas en función de la afinidad y la distribución de la memoria en sistemas NUMA es esencial para optimizar el rendimiento del sistema y garantizar un uso eficiente de los recursos disponibles.

1.2. Objetivos del proyecto

El proyecto tiene como objetivo desarrollar una herramienta con las características necesarias para obtener información sobre el rendimiento de programas multihilo en sistemas NUMA. Esta herramienta se empleará para caracterizar las distribuciones que proporcionan el mayor rendimiento en programas multihilo simples.

1.2.1. Desarrollo de un benchmark

Se desarrollará un *benchmark* en C++ [18] que posibilite la realización de pruebas de rendimiento en arquitecturas NUMA, para programas con múltiples tareas en ejecución y diversos conjuntos de datos, ya sean privados o compartidos. Específicamente, este *benchmark* permitirá:

- Generar n hilos de ejecución paralela y asociar cada uno a un nodo específico.
- Establecer el conjunto de datos al que accederá durante la ejecución.
- Definir de manera sencilla las operaciones que se desean llevar a cabo sobre los datos.

1.2.2. Uso del *benchmark*

Se utilizará el *benchmark* desarrollado para llevar a cabo pruebas en el CT-NUMA1 del CiTIUS, específicamente se analizará el rendimiento de:

- Un único hilo de ejecución en los diversos nodos al acceder a datos privados.
- Dos hilos en ejecución paralela al acceder a datos privados.

- Dos hilos en ejecución paralela al acceder a datos compartidos.

Estas pruebas se realizarán para operaciones básicas de lectura, escritura y lectura-escritura. Además, se caracterizará y se buscará obtener la distribución más afín, es decir, aquella en la que la disposición de los hilos y los datos, así como el grado de solapamiento de los accesos, proporcionen los resultados más eficientes.

1.3. Hipótesis

Dado que los recursos dentro del servidor de pruebas son limitados y que, como ya se mencionó, en los sistemas NUMA los accesos a datos locales son más eficientes, se espera que:

- La localidad de los hilos y los datos tenga una relevancia significativa en el rendimiento. Cuando los hilos acceden a datos que se encuentran en su mismo nodo, se espera un rendimiento mayor que cuando se realizan accesos remotos.
- La cercanía de los hilos entre sí tenga especial relevancia en el rendimientos cuando estos acceden al mismo conjunto de datos compartidos.
- Cuando los hilos acceden a conjuntos de datos diferentes, la cercanía de estos penalizará el rendimiento por la competencia en el uso de recursos compartidos.

Capítulo 2

Estado del conocimiento del problema a abordar

Pese a la gran cantidad de *papers* y *benchmarks* centrados en la caracterización del rendimiento de sistemas NUMA y paralelos, existe una carencia en la caracterización de comportamientos básicos así como, en la existencia de programas flexibles que permitan afinar las pruebas para este tipo de estudios.

2.1. *Benchmarks* de medición existentes

Los *benchmarks* informáticos son programas que establecen pruebas estándar del rendimiento de un ordenador y el software que lo utilizan [14]. De esta manera, permiten cuantificar la eficiencia, en términos. por ejemplo, del tiempo de ejecución de una aplicación o característica específica de un ordenador. Así, posibilitan medir, predecir y comprender del rendimiento de un programa o conjunto de programas en un sistema informático [2].

Actualmente existen una gran cantidad de *benchmarks* orientados a las pruebas de ejecución paralela, los cuales abarcan una amplia gama de aplicaciones y plataformas. Estos varían en complejidad, lenguajes de programación que utilizan y áreas específicas de la ejecución paralela que evalúan.

Cada *benchmark* tiene sus propias fortalezas y debilidades, y la elección de uno sobre otro puede depender de numerosos factores, incluyendo las necesidades específicas del usuario, las características del sistema que se está probando, y el tipo de información que se busca obtener.

No obstante, entre todas las alternativas disponibles, NAS *Parallel Benchmarks* (NPB) [11], SPEC Omp2012 [12] y *STREAM* [13] destacan como tres de las herramientas más utilizadas y reconocidas en el campo de la ejecución paralela.

- NAS *Parallel Benchmarks* (NPB): estos *benchmarks*, desarrollados por la NASA, conforman un conjunto preseleccionado de pruebas cuyo objetivo

principal es evaluar el rendimiento de supercomputadores paralelos. La *suite* incluye pruebas en aplicaciones de dinámica de fluidos computacional, pruebas con mallas adaptativas no estructuradas, entrada/salida paralela, aplicaciones multi-zona, entre otras. Todas las implementaciones de referencia de NPB están disponibles en los modelos de programación MPI y OpenMP. Sin embargo, los tamaños de los problemas en los NPB están preestablecidos y, aunque en la página oficial se ofrece el código para realizar las modificaciones que se consideren necesarias sobre las pruebas, este es bastante complejo. Por lo tanto, llevar a cabo ampliaciones o adaptaciones requeriría de una curva de aprendizaje significativamente alta [11].

- SPEC Omp2012: este *benchmark* está disponible de forma gratuita bajo licencia educativa. Está diseñado específicamente para ejecutar pruebas de programación paralela mediante OpenMP y dispone de un sitio web oficial que proporciona una gran cantidad de información sobre su uso. Una vez descargado, el paquete incluye el código fuente de los *benchmarks*, de manera que los usuarios pueden personalizarlos de acuerdo a sus requerimientos específicos. Además, incorpora las herramientas necesarias para compilar, ejecutar y generar informes sobre las pruebas.

  Sin embargo, fuera del ámbito educativo, el acceso al *benchmark* requiere el pago de una licencia. A pesar de que el código fuente puede ser modificado, dichas modificaciones no resultan sencillas, dado que las pruebas están específicamente diseñadas para actuar como estándares, de manera que puedan compararse con los resultados de otros usuarios de SPEC. Por último, las distintas pruebas están escritas en diferentes lenguajes de programación, incluyendo Fortran, C y C++. Por lo tanto, si un usuario desea realizar modificaciones, requerirá un conocimiento profundo de todos estos lenguajes [12].

- STREAM: este *benchmark*, desarrollado por el Departamento de Ciencias de la Computación de la Universidad de Virginia, es considerado el estándar de facto para medir el ancho de banda sostenido de computadores, según indica su propia página web. STREAM cuenta con una versión paralela en C que utiliza OpenMP para distribuir los *arrays* a los que se accede entre todos los hilos creados. Sin embargo, no permite modificar el tamaño de los *arrays* sin recompilar el código, ni definir de manera sencilla la ubicación de los hilos o *arrays* en los nodos de una distribución NUMA [13].

De este modo, al igual que los anteriormente mencionados, la mayoría de los *benchmarks* disponibles actualmente se enfocan en realizar pruebas preestablecidas difíciles de modificar. Así, ofrecen pruebas preconfiguradas que, aunque en algunas etapas pueden ser modificadas, no resulta sencillo hacerlo ya que no constituye su objetivo principal. Además, en general la disposición de hilos o datos en

nodos o *cores* especificados por el usuario, no resulta simple. En este contexto, surge la necesidad de contar con un software sencillo, que utilice un único lenguaje de programación y que permita modificaciones de manera fácil y directa. Esto posibilitaría la creación y definición de pruebas personalizadas por el propio usuario, adaptadas a sus necesidades específicas. En concreto, se considera conveniente este tipo de *software* para realizar pruebas en arquitecturas NUMA, ya que en su mayoría los *benchmarks* existentes se centran en realizar pruebas de ejecución paralela genéricas.

2.2. Estudios de rendimiento de sistemas paralelos y arquitecturas NUMA

Existen multitud de estudios que caracterizan el comportamiento de sistemas NUMA bajo distintas casuísticas. En concreto, tres de los más destacados son:

- *Thread and Memory Placement on NUMA Systems: Asymmetry Matters*: este *paper* se centra en mostrar los efectos sobre el rendimiento de las distintas disposiciones de hilos y datos sobre sistemas NUMA con interconexiones asímetricas, donde existen diferencias entre el ancho de banda de las distintas conexiones. En concreto muestra como, la asimetrías en las interconexiones en sistemas NUMA modernos tienen un impacto drástico en el rendimiento.

  En este sentido, se demuestra que el rendimiento se ve más afectado por el ancho de banda disponible entre nodos que por la distancia física entre ellos. Además, expone el desarrollo de un *scheduler* llamado *AsymSched*, que permite ajustar el posicionamiento de hilos y datos para maximizar el ancho de banda en la comunicación de hilos.

  Por último, se menciona cómo, a medida que el número de nodos incrementa en estos sistemas, resulta más complejo mantener simétricas las interconexiones, por los que los principios expuestos por el algoritmo tendrán una importancia creciente en el futuro de los NUMA [24].

- *Modeling memory bandwidth patterns on NUMA machines with performance counters*: se centra en presentar y evaluar un modelo que permite determinar los requerimientos de ancho de banda de programas multihilo en sistemas NUMA, en función de la distribución de hilos sobre los nodos. En concreto, parametriza el comportamiento de aplicaciones sintéticas, donde se posee un gran control de posicionamiento de los datos, y aplicaciones diseñadas para imitar comportamientos típicos de ejecuciones del mundo real. Estos resultados se contrastan con las predicciones del modelo, para estudiar su fiabilidad.

Tras realizar alrededor de 1000 experimentos, se muestra cómo el modelo presenta un alto grado de precisión para una amplia gama de aplicaciones con requerimientos de ancho de banda de moderando a alto. Además, se exponen técnicas para detectar cuando el modelo falla a la hora de ajustarse efectivamente a los resultados.

Las ideas mostradas en este *paper* tienen aplicaciones en sistemas de *debugeado*, herramientas de calendarización de tareas como Pandia [26] y librerías que controlan el posicionamiento de datos en memoria [25].

- *Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems*: muestra la relevancia en la actualidad de los efectos de la congestión de los recursos en sistemas NUMA, en concreto de los controladores de memoria e interconexiones cuando se ejecutan aplicaciones intensivas en accesos a datos.

  En particular, en el *paper* se presenta *Carrefour*, un algoritmo para sistemas NUMA que gestiona el tráfico de datos en los controladores de memoria e interconexiones. Se expone cómo otras políticas de manejo de memoria para estos sistemas se centran en mitigar el coste de accesos remotos, pese a que esta problemática ha ido perdido relevancia en los sistemas modernos. En contraposición, *Carrefour* se centra en los efectos provocados por la congestión de los recursos, haciendo hincapié en la relevancia que el algoritmo tendrá en un futuro no muy lejano, al preveerse una reducción del ancho de banda por *core* aumentando la relevancia de la congestión en el rendimiento [27].

En contraposición, este trabajo pretende aportar una base sistemática que modele comportamientos básicos con dos hilos e ilustre la variación en el rendimiento de los sistemas ante la compartición de recursos, la ejecución de hilos independientes concurrentes y/o el solapamiento de hilos paralelos con datos compartidos. Específicamente, las pruebas se enfocarán en establecer qué distribución es más "afín" en tres escenarios diferentes: el primero, al utilizar un solo hilo de ejecución que accede a datos privados; el segundo, al tener dos hilos trabajando en paralelo, cada uno con acceso a su propio conjunto de datos privados; y el tercero, donde dos hilos acceden a un conjunto compartido de datos, variando, en este último caso, el nivel de solapamiento en los accesos.

De esta manera, se pretende que el estudio modele situaciones comunes al ubicar hilos y datos en sistemas NUMA, esperando obtener unos resultados representativos de su rendimiento en un uso real.

Capítulo 3

Metodología

En este capítulo se caracterizará el servidor del Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS) sobre el que se llevarán las pruebas. Además, se expondrán las características generales de las pruebas que permiten evitar en la medida de lo posible la localidad espacial en los accesos, la precarga de datos y las optimizaciones del compilador.

E

3.1. Arquitectura del servidor empleado

Para la realización de las pruebas se utilizó un servidor NUMA conformado por 4 nodos con un Intel Xeon cada uno, distribuidos en forma de anillo.

3.1.1. Características técnicas de los nodos

Cada uno de los 4 nodos NUMA posee un procesador Intel Xeon CPU E5-4620 v4 con las características indicadas en la tabla 3.1. Para obtener resultados consistentes y estables la frecuencia de reloj de los núcleos se ha fijado en 2,1 GHz.

Arquitectura:	x86_64t	CPU MHz:	2100
L1d cache:	32KiB	CPU máx MHz:	2100
L2 cache:	256KiB	CPU mín MHz:	1200
L3 cache:	25600KiB		
Hilos por *core*:	1		
Cores por *socket*:	10		

Tabla 3.1: Características técnicas de cada procesador.

3.1.2. Topología de los Nodos

Los 4 nodos NUMA están distribuidos en forma de anillo, de forma que existen dos conexiones físicas por cada uno que lo conectan a otros dos de la red. Esta es una red de interconexión dedicada de ancho de banda homogéneo, de forma que las conexiones entre cualquier par de nodos son iguales. En la figura 3.1 se muestran la topología y la distancia entre nodos reportada por `numactl` [19] y en las figuras 3.2 y 3.3 se muestran latencias y ancho de banda medido entre los todos los pares de nodos (N).

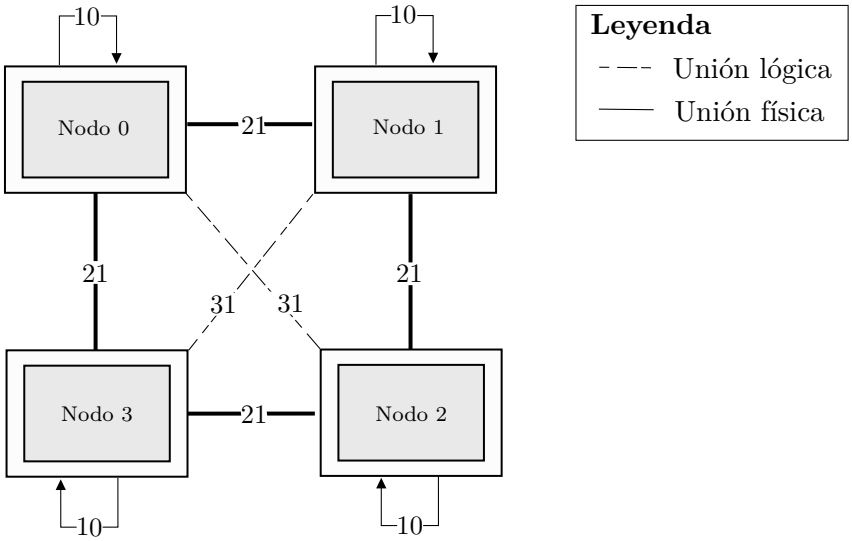


Figura 3.1: Topología del servidor donde se realizarán las pruebas [1].

N	1	2	3	4
0	87,7	254,2	271,2	255,4
1	254,9	86,0	253,2	271,5
2	271,4	252,7	86,0	254,7
3	255,1	271,9	254,3	85,7

Tabla 3.2: Latencias entre nodos (s).

N	1	2	3	4
0	61255	12709	12002	12368
1	12153	61176	12210	12007
2	12028	12395	61288	12689
3	12371	11992	12704	61261

Tabla 3.3: Ancho de banda entre nodos (MB/s).

3.1.3. Protocolo de coherencia caché

La red de interconexión emplea la tecnología *Intel QuickPath Interconnects* (QPI). Cada uno de los puertos QPI soportan dos conexiones unidireccionales para permitir comunicación bidireccional (*full-duplex*) entre dos componentes (nodos), de manera que se soporta el tráfico en ambas direcciones simultánea-

mente[1]. QPI define un modelo completo para la comunicación compuesto por cinco capas, representas en la figura 3.2. De estas cinco, de cara al estudio es especialmente relevante la capa de protocolo. Además, QPI emplea agentes que serán los encargados de realizar todas las acciones de coherencia caché [1].

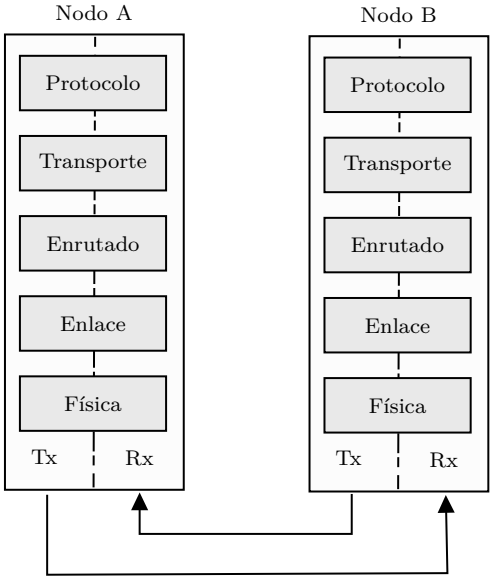


Figura 3.2: Modelo de 5 capas de QPI [1].

Tipos de agentes QPI emplea dos tipos distintos de agentes, *caching agents* y *home agents*:

- *Caching agent*: representa una entidad que puede iniciar transacciones de coherencia de memoria, está definido por los mensajes que transmite y su uso depende del comportamiento del protocolo de coherencia caché. Se encarga de:
  - Mantener copias de su propia estructura de caché.
  - Proveer copias de su memoria caché coherente a otros agentes (*Forward*).
  - Realizar solicitudes de lineas caché a otros agentes.
- *Home agent*: representa una entidad que ofrece transacciones coherentes, incluyendo el *handshaking*, en caso de ser necesario, con los *caching agents*. Se encargan de:

[1]En la figura 3.2 se representa los enlaces de recepción (Rx) y transmisión (Tx) asociados a una misma conexión.

- Supervisar una porción de la memoria coherente.
- Tratar los conflictos que pueden ocasionarse entre los distintos *caching agents*.
- Aporta respuestas de datos e información de propiedad según van siendo requeridas por un flujo de transacciones.

Los agentes tienen una lógica de funcionamiento que no está sujeta a los circuitos controladores de la memoria principal sino a la especificada por QPI, que mantiene la coherencia para un espacio de direcciones dado.

Capa de protocolo En esta capa los paquetes se definen como la unidad de transmisión. Estos se categorizan en seis clases diferentes: *home*, *snoop*, *data response*, *non-data response*, *non-coherent standard* y *non-coherent bypass*. Las peticiones y respuestas afectan a la coherencia del espacio de memoria o se utilizan para operaciones que no requieren de coherencia (configuración, mapeado de memoria de entrada/salida, interrupciones y mensajes entre agentes).

La coherencia de todas las cachés distribuidas en los nodos es mantenida por los agentes explicados anteriormente, sujetos a las reglas definidas por esta capa. El protocolo QPI permite dos modos de funcionamiento *home snoop* y *source snoop*[2]:

- *Home snoop*: está optimizado para permitir gran escalabilidad y se utiliza para sistemas grandes donde existe mayor tráfico de *snoop*.
- *Source snooop*: está optimizado para ofrecer menor latencia y se utiliza principalmente en sistemas más pequeños donde el número de agentes crea una cantidad de trabajo de *snoop* relativamente bajo.

Estos comportamientos definen el flujo de mensajes que se intercambiarán los agentes para garantizar la coherencia sobre las lineas caché. Además de este flujo de información, QPI expone distintos estados en los que una línea caché se puede encontrar en un momento dado. En concreto, QPI implementa una modificación del protocolo MESI, donde todas las lineas cachés están en uno de los estados: *modified* (M), *exclusive* (E), *shared* (S) o *invalid* (I), introduciendo el estado *Forward* (F) de reenvío de solo lectura para permitir el reenvío de líneas limpias de caché a caché. Solo un agente puede tener una línea caché concreta en este estado *Forward* en un momento específico, el resto de agentes pueden tener copias en estado *Shared* [20, 1, 21]. Las características de este protocolo están resumidas en la figura 3.4 donde:

- Limpia/sucia: indica si se ha modificado el dato en caché tras haberse leído.

[2]Los nombres de estos comportamientos indican qué agente realizarán el *snoop*, pudiendo ser el solicitante de la línea (*source*) o el encargado de gestionar ese segmento de memoria (*home*).

- ¿Puede escribirse?: indica si es posible escribir la línea caché sin necesidad de enviar un mensaje a algún otro agente del sistema.
- ¿Puede *forward*?: indica si se permite el envío directo de esa línea caché a otro nodo.
- Puede transitar a: indica qué está permitido que un agente haga con su estructura interna de la caché sin tener que enviar ningún mensaje a otro de los agentes del sistema, cualquier otra transición (como una invalidación de una línea *Modified*) requeriría intercambiar los mensajes de coherencia pertinentes [22].

Estado	**Limpia/sucia**	**¿Puede escribirse?**	**¿Puede *forward*?**	**Puede transitar a**
M	Sucia	Sí	Sí	—
E	Limpia	Sí	Sí	MSIF
S	Limpia	No	No	I
I	—	No	No	—
F	Limpia	No	Sí	SI

Tabla 3.4: Resumen del funcionamiento de MESIF [22].

Comportamientos de coherencia de QPI Como ya se mencionó, existen dos modos de funcionamiento que organizan los flujos de datos que mantienen la coherencia caché. A continuación se expone, para ambas configuraciones, un ejemplo de lectura desde el nodo 0 cuando los datos se encuentran en el nodo 2 en estado M, E o F:

- *Home Snoop*: en este modo de funcionamiento se define al *Home agent* como el responsable de realizar el *snooping* a los otros *caching agents*. El comportamiento típico involucra hasta 4 pasos.
  1. El *caching agent* (nodo 0) manda una petición al *home agent* (nodo 2) que se encarga de gestionar la línea de memoria a la que se quiere acceder[3].
  2. El *home agent* (nodo 2) emplea un directorio propio para dirigir un *snoop* al *caching agent* que puede que tenga una copia del fragmento de memoria solicitado en cuestión.

[3]Que un *home angent* se encargue de gestionar una dirección de memoria no significa que se este se encuentre en el nodo/procesador que tenga el fragmento de memoria asociado a su dirección en su caché.

3. El *caching agent* responde al *home agent* (nodo 0) con el estado de la dirección de memoria. En este caso, como la linea está en el estado M, E o F, se envían los datos al *caching agent* solicitante original (nodo 0) indicando al *home agent* (nodo 2) que los datos han sido enviados.
4. El *home agent* (nodo 2) resuelve cualquier conflicto que suceda durante la transacción, envía los datos al *caching agent* solicitante (nodo 0) (en caso de ser necesario) y envía un mensaje de transacción completada al finalizar.

- *Source Snoop*: este comportamiento simplifica la ejecución de una transacción, permitiendo al *caching agent* que solicita los datos enviar la petición al *home agent* y los propios mensajes de *snoop*. El comportamiento básico involucra hasta 3 pasos.

  1. El *caching agent* (nodo 0) envía una petición al *home agent* (nodo 2) que maneja el fragmento de memoria en cuestión y envía mensajes de *snoop* al resto de *caching agent* para comprobar si poseen copias de esos datos.
  2. Los *caching agent* responden al *home agent* (nodo 0) con el estatus de la dirección de memoria consultada. En este ejemplo, como el nodo 2 tiene una copia de la linea en estado M, E o F puede realizar un *forward*, por lo que envía los datos directamente al *caching agent* solicitante asociado al nodo 0.
  3. El *home agent* resuelve cualquier conflicto que pueda surgir e informa al solicitante (nodo 0) de que ha finalizado la transacción[4].

Source snoop ofrece el flujo más inmediato, permitiendo que el nodo 0 obtengan la línea en el segundo paso. Sin embargo, esta mejora es a costa de una mayor cantidad de mensajes entre agentes en la red de interconexión, ya que los *caching agent* deben realizar un *broadcast* con cada mensaje QPI que envíen. Es por ello que, actualmente versiones posteriores a QPI como UPI únicamente permiten un modo de funcionamiento basado en directorios [23]. De esta forma, con el objetivo de que los resultados obtenidos en las pruebas puedan reflejar mejor el comportamiento de versiones más actuales del protocolo, el servidor se configuró en modo *home snoop*.

3.2. Características de los códigos a ejecutar

Con el fin de facilitar la reproducibilidad de los experimentos y caracterizar completamente el coste de acceso a memoria, en todas las pruebas:

[4] Aunque la línea ya ha sido enviada directamente desde el nodo 2, es necesario que el *home agent* responda con un mensaje de finalización de la transacción para la correcta liberación de recursos.

- Se empleará un *array* para el almacenamiento de los datos y no un objeto tipo `std::vector`, al observarse en este durante estudios preliminares un sobrecoste en los accesos asociado.
- Se evitará el almacenamiento de datos en caché: se recorrerá un *array* de `char` de manera que el número de elementos coincida con el número de *bytes* que ocupa. En concreto, el vector tendrá un tamaño de 209,7152 MB, que corresponde con 8 veces la capacidad máxima de la caché L3 (ver tabla 3.1), de forma que el vector no pueda residir completamente en caché.
- Se evitará la localidad y la precarga: el *array* de elementos se recorrerá introduciendo un *stride* de 192 B, evitando la localidad espacial de datos presentes en una misma línea caché (64 B) y la precarga de las siguientes dos líneas consecutivas.
- Se evitarán las optimizaciones del compilador.
  - Empleando bucles `while` para recorrer el vector al ser menos probable la optimización de estos bucles que de los bucles `for`.
  - Compilando el código con la opción `-O0` de `gcc`, evitando que se realicen la mayor parte de optimizaciones del compilador [28].
  - Usando el *qualifier* `volatile` en las variables empleadas para iterar y para realizar las operaciones de acceso, evitando reordenaciones u optimizaciones por parte el compilador que dificulten el estudio [29].
  - Empleando una variable auxiliar para realizar todos los accesos y aplicando sobre ella la función `doNotOptimizeAway`, para evitar optimizaciones sobre el acceso a los datos [30].
- Se realizarán diez ejecuciones de cada experimento y se calculará la media del tiempo de ejecución.

De esta forma, se recorrerá el vector empleando la estructura de función expuesta en el código 3.1, sustituyendo el comentario del bucle más interno por el tipo de acceso a ejecutar.

```
int accesos(const vectorSize size){
    volatile vectorSize j = 0, i = 0;
    volatile int add = 0;

    while(j < dist){
        i = j;
        while(i < size){
            //Operacion a ejecutar
            i += dist;
        }
        j++;
    }
    return add;
}
```

Código 3.1: función para iterar sobre el array.

Capítulo 4

Pruebas

En este capítulo se caracterizarán todas las pruebas realizadas en el trabajo indicando: los tipos de operaciones, las características propias de cada experimento, los objetivos que pretenden alcanzar y la hipótesis de resultados esperados.

4.1. Operaciones a ejecutar

Los experimentos realizados en este trabajo, se enfocarán en caracterizar de manera exhaustiva los comportamientos más elementales de acceso a memoria, lecturas (R), escrituras (W) y lecturas-escrituras (R-W). Las operaciones se realizarán tal y como se muestran en los códigos 4.1, 4.2 y 4.3

```
1 add = vector[i];
```

Código 4.1: lecturas.

```
1 add = 'X';
2 vector[i] = add;
```

Código 4.2: escrituras.

```
1 add = vector[i];
2 vector[i] = add + 1;
```

Código 4.3: lectura-escritura.

4.2. Definición de pruebas, objetivos e hipótesis

Este trabajo tiene como objetivo aportar una base sistemática que modele el comportamiento de programas con uno y dos hilos, en función de la distribución de estos con respecto a los datos a los que acceden. Así, se pretende caracterizar completamente comportamientos básicos con formas de acceso típicas de manera que, los resultados obtenidos sirvan de referencia para investigaciones futuras.

Específicamente, las pruebas se enfocarán en establecer qué distribución de hilos y datos sobre los nodos es más "afín" en tres escenarios diferentes. A continuación se exponen las características de cada prueba, el objetivo que pretende alcanzar, el método concreto empleado para su ejecución con información específica de los datos que se van a recabar y las hipótesis de resultados esperados.

4.2.1. Un hilo y datos privados

Esta prueba medirá el coste de acceso de un único hilo, a su conjunto de datos privados, cuando no existen otros hilos concurrentes en ejecución que puedan afectar a su rendimiento.

Objetivo

Caracterizar el coste de acceso base de un hilo a sus datos en las distintas distribuciones posibles del hilo y los datos en los nodos. Este coste base hace referencia al tiempo de ejecución de un hilo cuando no existen otros hilos en ejecución y por lo tanto no existen, en primera instancia, problemas de contención de recursos[1] o de coherencia caché.

Método

Se definirá una matriz M de dimensiones $N \times N$, donde N corresponderá con el número de nodos de ejecución disponibles en el servidor. Las filas indicarán el nodo donde están ubicados los datos y la columna el nodo donde se encuentra el hilo. De esta manera, el elemento M_{ij} corresponderá con el coste de acceso en segundos cuando los datos se encuentran en el nodo i y el hilo en el nodo j.

Hipótesis

Tal y como se expuso en la sección 3.1.2, la red de interconexión es bidireccional y de ancho de banda homogéneo. De esta manera se espera que:

1. Las matrices resultantes sean simétricas, de forma que el coste de acceso de un hilo desde el nodo 0 a un conjunto de datos en el nodo 1, sea prácticamente el mismo que el coste de acceso de un hilo en el nodo 1 a un conjunto de datos en el nodo 0.

2. El coste de acceso a los datos se incremente a medida que estos se encuentren más distanciados, siguiendo la topología expuesta en la figura 3.1.

[1] La contención hace referencia a los conflictos producidos cuando varios hilos, procesos, máquinas... intentan acceder y ocupar un recurso compartido simultáneamente ya sean memoria principal, almacenamiento de disco, caché, buses internos etc [31].

4.2.2. Dos hilos y datos privados

En esta prueba se ejecutarán dos hilos en paralelo, que accederán cada uno a su conjunto de datos privados.

Objetivo

Caracterizar el efecto que tiene la contención y compartición de recursos sin incorporar sobrecostes asociados a los mecanismos de coherencia caché. En concreto los recursos a estudiar serán: el bus y controlador de memoria, el bus QPI, y el procesador asociado a cada nodo.

Método

En función de los resultados de las pruebas con *un hilo y datos privados*, se seleccionará una de las distribuciones con mayor coste temporal. De esta manera, se pretende acentuar la diferencia de coste entre los mejores y peores resultados.

Dado que existen 16 formas posibles de distribuir dos parejas de hilos y datos entre dos nodos, nos enfocaremos en aquellas consideradas más representativas. Estas configuraciones englobará todas las combinaciones de uso de los tres recursos a compartir (bus QPI (BQ), bus de memoria y controlador (MC), y procesador (S)), con excepción de las configuraciones en las que se comparte únicamente el bus de memoria y el bus QPI, o el bus QPI y el procesador, al no ser posibles en el problema enmarcado.

En concreto obtenemos 6 posibles situaciones, de compartición de recursos:

- Solo S.
- Solo BQ.
- Solo MC.
- MC y S.
- MC, BQ y S.
- Ninguno.

Para todas las distribuciones y las distintas operaciones de lectura, escritura y lectura-escritura, se comparará:

- El recurso que es compartido con los dos hilos en cada distribución.
- El coste, como la suma del tiempo de ejecución de ambos hilos en segundos. En lugar de escoger el hilo de mayor duración se calcula la suma del coste de los hilos, de forma que el resultado se vea afectado por la aportación de ambos hilos de cara al estudio.
- El *overhead*, como el incremento del coste de ejecución de cada distribución para una operación (lectura, escritura y lectura-escritura) con respecto al coste que se obtendría si ambos hilos se ejecutasen de manera aislada. Es decir, para cada distribución y cada operación se calculará:

$$overhead(\,\%) = \left(1 - \frac{b_0 + b_1}{n_0 + n_1}\right) \cdot 100. \tag{4.1}$$

Donde:

- b_k: es el coste que tendría el hilo k al acceder a sus datos si no se encontrase en ejecución el otro hilo, según los resultados observados en la prueba con *un hilo y datos privados*.
- n_k: es el nuevo coste de ejecución obtenido para el hilo k.

Hipótesis

Es de esperar que el rendimiento se deteriore a medida que se comparten más recursos, en especial en los casos del bus y controlador de memoria y, el bus QPI, dado que ambos hilos no pueden estar transmitiendo datos simultáneamente por estos canales. Por otro lado, debido a que las pruebas son intensivas en accesos a memoria y no en cálculo y que cada hilo se ejecutará en un *core*, la compartición del procesador debería tener un efecto menor.

4.2.3. Dos hilos y datos compartidos

En este caso, las pruebas se realizarán sobre dos hilos que acceden en paralelo a un mismo conjunto de datos compartido.

Objetivo

Caracterizar los efectos que el protocolo de coherencia caché pueda tener en el tiempo de ejecución, en función del grado de solapamiento del acceso de los hilos a los datos.

Método

Dado que debemos distribuir los dos hilos y un conjunto de datos entre dos nodos, existen 8 combinaciones posibles. Sin embargo, nos centraremos en los efectos asociados al acceso remoto o local de cada hilo a los datos. De esta forma, obtenemos tres distribuciones

- Local-Local (LL): ambos hilos tienen acceso a los datos de forma local.
- Local-Remoto (LR): uno hilo accede a los datos de forma local y otro de forma remota a través del canal QPI.
- Remoto-Remoto (RR): ambos hilos acceden a los datos de forma remota a través del canal QPI.

Estas pruebas se repetirán con distinto grado de solapamiento en el acceso, en concreto:

- Accesos solapados: ambos hilos accediendo al vector de forma simultánea desde el mismo elemento de partida.
- Accesos solapados con reducción de trabajo: ambos hilos accederán de forma simultánea y desde el mismo elemento, pero uno de ellos procesará un conjunto menor de datos.
- Accesos semi-solapados: uno de los hilos incorporará un retardo en el acceso a los datos. En concreto, para caracterizar de forma clara este comportamiento, estudiaremos la evolución del tiempo de ejecución de los hilos a medida que incrementamos el retardo de acceso de uno de ellos al vector.

Hipótesis

Se espera que el protocolo de coherencia caché tenga especial relevancia en las operaciones que involucran escrituras, debido al aumento de los conflictos entre los dos hilos. Además, a medida que se reduce el grado de solapamiento, se espera una reducción del coste de ejecución.

Capítulo 5

Discusión de los resultados

En este capítulo se exponen y discuten los resultados obtenidos para las pruebas descritas, además de realizarse conjuntos de experimentos auxiliares para contrastar resultados en caso de ser necesario.

5.1. Un hilo y datos privados

En las tablas 5.1, 5.2 y 5.3 observamos los costes de accesos asociados a lecturas, escrituras y lecturas-escrituras respectivamente.

Vemos cómo se confirman los dos comportamiento esperados:

1. En todos los casos se observa un comportamiento simétrico con respecto a la diagonal. Este comportamiento concuerda con lo estudiado en la sección 3.1.2, dado que las conexiones entre nodos tienen un ancho de banda homogéneo y son bidireccionales.

2. El coste de acceso crece a medida que incrementa la distancia entre los datos y el hilo, siguiendo la topología de la figura 3.1. De esta forma, los accesos más veloces suceden cuando los datos y el hilo se encuentran en el mismo nodo, seguidos de los accesos a datos que se encuentran en nodos adyacentes al del hilo y, por último, los accesos que requieren atravesar un nodo intermedio.

De esta manera, observamos cómo las disposiciones más afines para un hilo que accede a sus datos privados sin la presencia de otros hilos concurrentes, son aquellas en las que los hilos y los datos se encuentran en el mismo nodo.

Es interesante destacar el coste temporal asociado a las escrituras. Estas, muestran un coste de acceso hasta 9 veces superiores a las lecturas y, casi 7 veces superiores a las operaciones de lectura-escritura. A continuación, en la sección 5.1.1, se muestran dos grupos de pruebas diseñados para determinar la naturaleza de estos comportamientos.

		Hilo			
		N0	**N1**	**N2**	**N3**
Datos	**N0**	2,0307	5,8903	6,0408	5,8923
	N1	5,9153	2,0355	5,9121	6,1039
	N2	6,0733	5,9569	2,0281	5,9095
	N3	5,9467	6,0511	6,0511	2,0253

Tabla 5.1: Costes de lecturas (segundos).

		Hilo			
		N0	**N1**	**N2**	**N3**
Datos	**N0**	19,1069	31,0287	40,3092	30,6787
	N1	30,7297	19,0377	30,6631	40,3776
	N2	40,3638	30,6935	18,6312	30,5306
	N3	30,7069	40,0940	30,8610	18,9247

Tabla 5.2: Costes de escrituras (segundos).

		Hilo			
		N0	**N1**	**N2**	**N3**
Datos	**N0**	3,3372	5,0028	6,3672	4,9907
	N1	4,9637	3,3016	4,9850	6,3658
	N2	6,3785	4,9816	3,2891	4,9684
	N3	5,0374	6,3541	5,0072	3,3304

Tabla 5.3: Costes de lectura-escritura (segundos).

5.1.1. Estudio del comportamiento de las operaciones de escritura

En un primer momento se podría esperar que el comportamiento en las escrituras fuese *write-allocate*. De esta manera, los fallos de escritura se manejarían como un fallo de lectura, trayendo el dato de memoria a caché y realizando la operación de escritura sobre caché [34]. Así, las pruebas de escritura deberían ofrecer unos resultados muy similares a las lecturas-escrituras.

Sin embargo, tal y como se observó, los costes de realizar escrituras aisladas resultan en ejecuciones varias veces más lentas que realizar operaciones de lectura-escritura. Este comportamiento podría ser explicado por múltiples factores, en concreto, estudiaremos dos hipótesis, comportamientos asociados a precargas no evitadas en las pruebas y un posible *bypass* de la caché.

Efectos de la precarga

Esta hipótesis se basa en tres componentes; primero, la política de escritura sí se corresponde con *write-allocate*; segundo, a la hora de realizar las pruebas no se ha evitado completamente la precarga de líneas caché; y tercero, la precarga es asimétrica entre lecturas y escrituras, siendo mayor en la primera, al considerarse más crítica en términos de localidad.

De esta forma, las lecturas estarían moviendo de memoria a caché más *bytes* que el *stride* introducido en las pruebas (192 B), incurriendo en aciertos caché en la lectura de algunos elementos del *array*. Sin embargo, las escrituras estarían realizando un *prefetch* menor, de manera que sus accesos implicarían un número mayor de fallos caché. Así, al realizar operaciones de lectura-escritura se reduciría el número de fallos caché, al aprovechar la precarga de las lecturas, resultando en un menor tiempo de ejecución.

El comportamiento de precarga se pueden comprobar de forma simple, incrementando el número de bytes de *stride* entre accesos de lecturas y escrituras. En la figura 5.1 vemos que, tanto para las lecturas como para las escrituras, el rendimiento posee una tendencia de aumento del tiempo de ejecución, que se vuelve estable al alcanzar un *stride* de 192 B. Así, observamos como, por un lado, se estaba evitando correctamente la precarga y, por otro, que las escrituras aisladas, son intrínsecamente peores que las lecturas al incrementar el *stride*. Por lo tanto, descartamos esta hipótesis para el comportamiento de las escrituras.

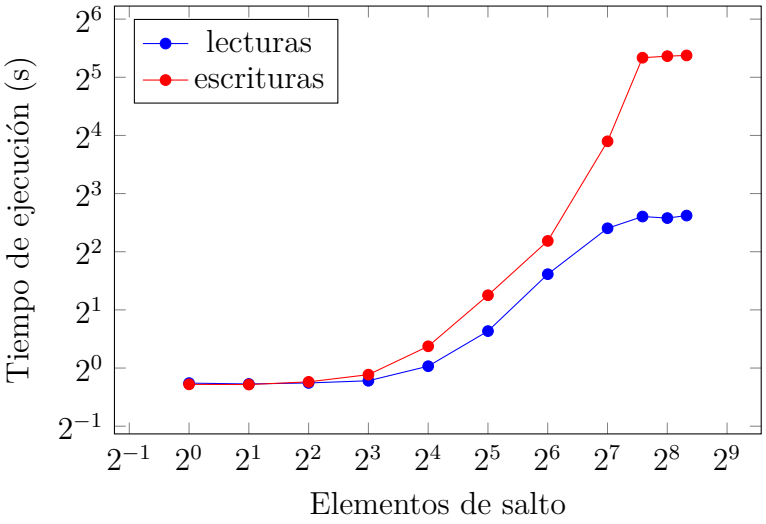


Figura 5.1: Tiempo de ejecución en lectura y escrituras en función del *stride*.

Bypass de la caché en escrituras

En este caso, la hipótesis que explica el comportamiento se basa en la realización de escrituras directamente en memoria, haciendo un *bypass* de la caché.

En concreto se basa en tres componentes; primero, al no reutilizarse el dato escrito en el código, no existe localidad temporal, por lo que no se trae el dato a caché, escribiéndose directamente en memoria; segundo, las escrituras en memoria principal son, intrínsecamente, más lentas que las lecturas; y tercero, existe un buffer intermedio de escritura en memoria. De esta manera el sistemas se comportaría de las siguiente forma:

Política *no-write-allocate* En este caso cuando se realiza una escritura y el dato no se encuentra en caché, se escribe directamente en memoria [34]. Este tipo de comportamiento se suele emplear cuando, los datos escritos no son consumidos "pronto"[1]. En estas situaciones, cargar una línea caché en memoria únicamente para sobreescribirla, sin reutilizar el dato, resulta en un deterioro del rendimiento. Esto es debido a que, podría ocupar la posición de un línea que sí va a ser reutilizada en la ejecución, generando un incremento de los fallos caché [35]. Este tipo de comportamiento puede ser en parte demostrado realizando un nuevo grupo de pruebas con operaciones de escritura-lectura. En la tabla 5.4 vemos cómo estas operaciones de escritura-lectura suponen un coste menor que las operaciones escritura aisladas. De esta manera, podemos observar comportamientos diferentes en función de la reutilización o no de los datos.

		Hilo			
		N0	**N1**	**N2**	**N3**
Datos	**N0**	3,8374	6,3264	7,8953	6,3534
	N1	6,3902	3,8465	6,3488	7,8974
	N2	7,9655	6,3824	3,7910	6,3081
	N3	6,3840	7,9333	6,3710	3,8432

Tabla 5.4: Costes de escritura-lectura (segundos).

Adicionalmente, es posible realizar un estudio del número de fallos caché empleando *Intel VTune Profiler* [36]. Los resultados muestran un total de 64 millones de fallos caché para operaciones de lectura, y 0 fallos para las escrituras. En consecuencia, se puede deducir que no se están cargando en caché los datos a escribir.

Escrituras más lentas que lecturas Como en las operaciones de lectura y escritura se están realizando accesos a memoria, según la explicación anterior, las escrituras deberían tener un coste mayor a las lecturas.

En memorias DDR (*double data rate*) aunque es posible realizar tanto escrituras como lecturas en la fase "alta" y "baja" del ciclo de reloj, estas están optimizadas para realizar lecturas [32], de forma que las escrituras son más lentas. Como se muestra en la figura 5.1, independientemente del *stride* introducido, las operaciones de escritura son más lentas.

[1]Esta definición de "pronto" dependerá del fabricante de la arquitectura.

***Buffer* de escritura** Si los datos se están escribiendo directamente en memoria, realizando un *bypass* de la caché, aún es necesario explicar por qué en la figura 5.1 las escrituras se ven afectas por la localidad espacial.

En este caso, esta reducción del rendimiento hasta realizar un *stride* de 192 B, estaría causado, no por la precarga de 3 líneas caché de 64 B, sino por la existencia de un buffer de escritura de 192 B.

Esta explicación encaja con la el uso de un *Write Combine Buffer* (WCB) nombrado en múltiples documentos de Intel [41, 42]. Cuando se produce un fallo de escritura en la L1, con el fin de evitar que la CPU tenga que esperar mientras se realiza la búsqueda en el resto de niveles de la jerarquía de memoria, el dato se escribe en un WCB [40]. Este *buffer* combina escrituras a direcciones de memoria consecutivas, de forma que se puedan realizar en ráfagas (*burst*) sobre memoria o caché.

Las escrituras se realizan en ráfagas debido a que, los controladores de memoria actuales priorizan las lecturas a las escrituras, retrasando lo máximo posible las segundas [37, 39]. Cuando finalmente es necesario realizar las escrituras, estas ráfagas permiten reducir efectos como el *bus turnaround delay*[2].

Podemos comprobar de forma simple esta situación en la figura 5.2. Si repetimos las pruebas empleando *chars* y *floats* vemos como el tiempo de ejecución se estabiliza en ambos casos en 192 B. En el caso de los *chars* el coste de ejecución se comienza a estabilizar al realizar saltos de 192 elementos. Al utilizar *floats* el coste se estabiliza al introducir un *stride* de 48 elementos, que coincide con un salto de 192 B ($48\,\textit{floats} \cdot 4\,\text{B}/\textit{float} = 192\,\text{B}$).

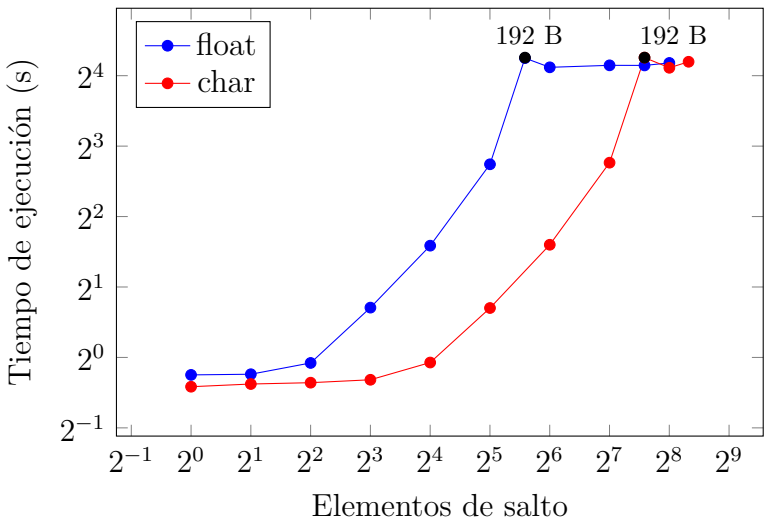


Figura 5.2: Relación salto-tiempo para *char* y *float*.

[2]El *bus turnaround delay* es el retardo producido cuando un bus cambia la dirección de transmisión, en este caso de lecturas memoria-CPU a escrituras CPU-memoria [38].

Vemos así cómo las escrituras se están realizando en ráfagas de 192 B consecutivos, cuando se introduce un *stride* mayor, el coste se estabiliza dado que en cada ráfaga se estará escribiendo un único elemento.

5.1.2. Recapitulación

En las pruebas se observó cómo la distribución más afín es aquella en la que los datos se encuentran locales al hilo. Además que, debido a la implementación interna de las operaciones de escritura, el coste de realizar escrituras aisladas es muy superior al de realizar operaciones de lectura, lectura-escritura y escritura-lectura. En concreto, se observó que este efecto se produce debido a que los datos se escriben realizando un *bypass* de la caché, haciéndose directamente en memoria, pasando por un *buffer* intermedio que realiza escrituras en ráfagas de datos contiguos.

5.2. Dos hilos y datos privados

Tal y como se mencionó en la sección 4.2.2, las pruebas se realizaron empleando los dos nodos que impliquen un acceso más lento entre ellos. En este caso, de las dos parejas existentes, se eligieron los nodos 0 y 2. En las tablas 5.5 a 5.10 se muestran los resultados de cada distribución y en las figuras 5.3 a 5.8 se muestra la configuración concreta de hilos y datos en los dos nodos.

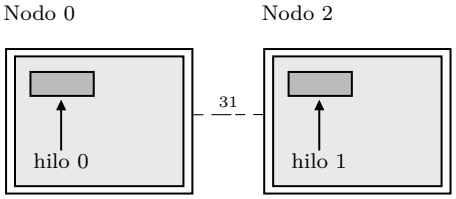


Figura 5.3: Diagrama ningún recurso compartido.

	R	W	R-W
Hilo 0	2,0448	19,1014	3,3380
Hilo 1	2,0344	19,0986	3,3175

Tabla 5.5: Resultados ningún recurso compartido (segundos).

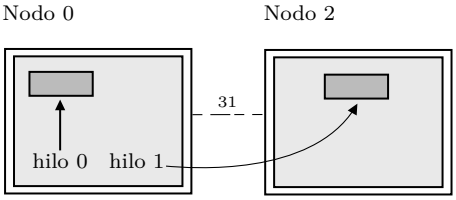


Figura 5.4: Diagrama solo S.

	R	W	R-W
Hilo 0	2,0364	19,0631	3,3036
Hilo 1	6,0881	40,3200	6,3538

Tabla 5.6: Resultados solo S (segundos).

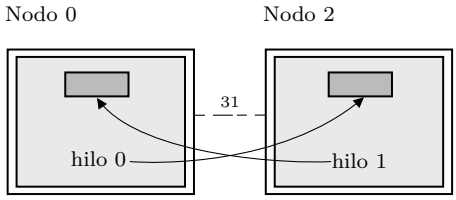


Figura 5.5: Diagrama solo BQ.

	R	W	R-W
Hilo 0	6,2586	40,6708	6,6771
Hilo 1	6,2119	40,4045	6,6658

Tabla 5.7: Resultados solo BQ (segundos).

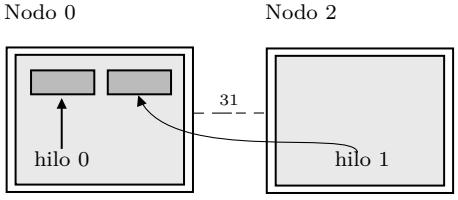


Figura 5.6: Diagrama solo MC.

	R	W	R-W
Hilo 0	2,1008	19,4652	3,4059
Hilo 1	6,0895	40,4284	6,4050

Tabla 5.8: Resultados solo MC (segundos).

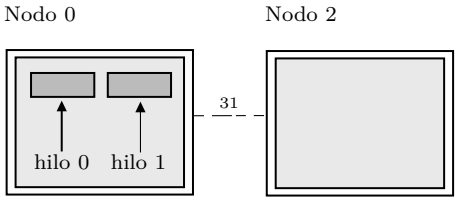


Figura 5.7: Diagrama MC y S.

	R	W	R-W
Hilo 0	2,1509	19,2046	3,4267
Hilo 1	2,1705	19,6006	3,4447

Tabla 5.9: Resultados MC y S (segundos).

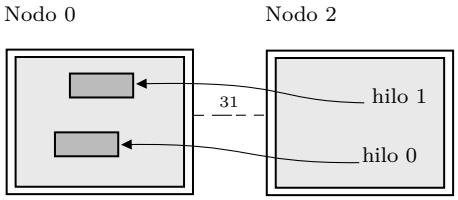


Figura 5.8: Diagrama MC, BQ y S.

	R	W	R-W
Hilo 0	6,9380	40,4259	7,2322
Hilo 1	6,9561	40,6677	7,2586

Tabla 5.10: Resultados MC, BQ y S (segundos).

En la tabla 5.11 observamos para las operaciones de lectura, escritura y lectura-escritura, el coste de cada distribución y el *overhead* (ver ecuación 4.1) con respecto a la versión con *un hilo y datos privados*.

Vemos cómo se confirman, en el caso de las lecturas y las lecturas-escrituras, los dos comportamientos esperados:

- El *overhead* aumenta al incrementar el número de recursos compartidos.
- El *overhead* asociado a la compartición del procesador es limitado, siendo inferior en todos los casos al 1 %.

En general, al compartir un solo recurso, el mayor *overhead* se observa al compartir el bus QPI o el bus de memoria y controlador. Por otro lado, el sobrecoste de la compartición del procesador es apreciable solo cuando se emplea además otro recurso compartido. Esta situación se observa por ejemplo en las lecturas al compartir el procesador y, el bus de memoria y controlador donde, el *overhead* pasa del 2,18 % a 10,66 %. De esta forma, el mayor *overhead* se obtiene al compartir todos los recursos, llegando al 14 %.

		Prueba asociada a la figura					
		5.3	**5.4**	**5.5**	**5.6**	**5.7**	**5.8**
Recurso compartido	**Bus de memoria y Controlador**				x	x	x
	Bus QPI			x			x
	Procesador		x			x	x
R	**Coste (segundos)**	4.0	8.1	12.5	8.2	4.5	14.2
	Overhead (%)	0.06	0.66	2.90	2.18	10.66	14.75
W	**Coste (segundos)**	37.7	59.5	80.9	59.8	39.1	81.0
	Overhead (%)	1.01	0.82	0.27	1.27	4.44	0.37
R-W	**Coste (segundos)**	6.5	9.6	13.4	9.8	6.9	14.8
	Overhead (%)	0	0	5.22	1.75	5.20	14.15

Tabla 5.11: Relación de la compartición de recursos con el coste y el *overhead*, para la distintas pruebas representadas en los diagramas 5.3 a 5.8.

Sin embargo, en el caso de las escrituras, el comportamiento no se ajusta a las hipótesis expuesta. En general, el *overhead* se mantiene en torno al 1 %, salvo en el caso de la compartición del bus de memoria y controlador, y procesador, donde aumenta hasta el 4 %.

No obstante, si repetimos los experimentos, observamos como los costes de lecturas y lecturas-escrituras se mantienen constantes, mientras que en el caso de las escrituras, existe una mayor variación, reduciéndose el *overhead* de la prueba 5, hasta el 2 %. Así, vemos que al realizar escrituras asiladas, donde se produce un *bypass* de la caché (ver sección 5.1.1), existe una mayor variación de los resultados, impidiendo obtener información concluyente sobre el efecto de la compartición de recursos.

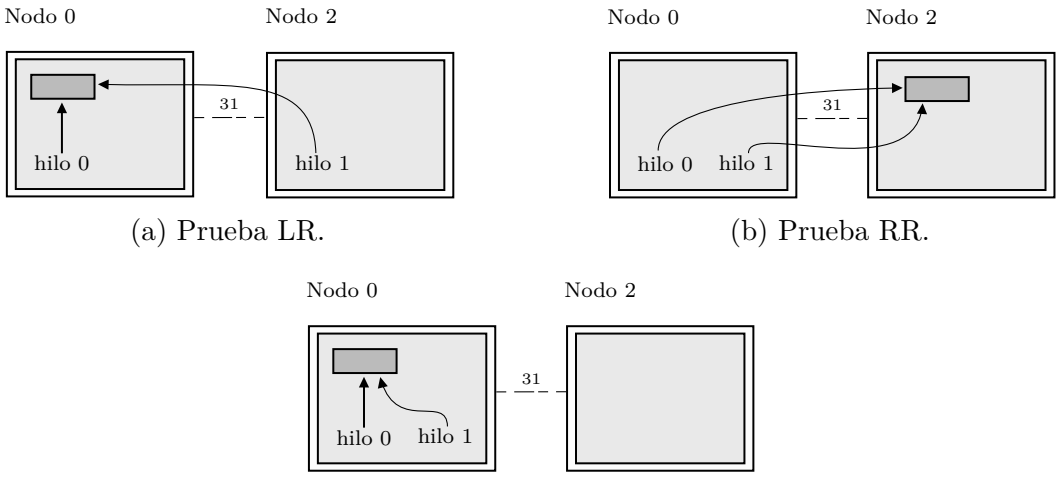
5.2.1. Recapitulación

En las pruebas se observó cómo, de nuevo, la distribución más afín es aquella en la que cada pareja de hilo y datos se encuentra en nodos independientes sin compartir recursos. En el caso de las operaciones de lectura y lectura-escritura cuando sí existe compartición de recursos, la mayor penalización al rendimiento se da al compartir el bus de memoria y controlador, y el bus QPI. Este efecto, alcanza su pico al compartir todos los recursos, llegando a producir un *overhead* de más del 14 %.

En el caso de las escrituras, los resultados son más variables debido a la forma en la que estas se realizan sobre memoria, por lo que no se puede observar un efecto directo achacable a la compartición de recursos.

5.3. Dos hilos y datos compartidos

En las figuras 5.9a 5.9b y 5.9c vemos las distintas distribuciones sobre las que se realizarán las pruebas.



(a) Prueba LR.

(b) Prueba RR.

(c) Prueba LL.

Figura 5.9: Diagramas de pruebas con dos hilos y un conjunto de datos.

5.3.1. Accesos solapados

En las tablas 5.12 5.13 y 5.14 observamos los costes de las 3 configuraciones (ver figura 5.9). Vemos que no se cumple la primera hipótesis de la sección 4.2.3. El protocolo de coherencia no solo tiene un efecto relevante en las escrituras, sino que también en operaciones de lectura y lectura-escritura. De esta forma, se observa una sincronización en todos los casos entre los dos hilos. En concreto resalta el comportamiento de la *prueba LR* (ver tabla 5.12) donde el hilo 0, que accede a los datos de forma local, obtiene unos resultados 3 veces peores a los obtenidos en la prueba con *un hilo y datos privados*. Así, el hilo local se llega a sincronizar hasta la diezmilésima de segundo con el remoto.

	R	**W**	**R-W**
Hilo 0	6,4941	40,3943	6,4832
Hilo 1	6,4941	40,4409	6,4997

Tabla 5.12: Resultados LR accesos solapados (segundos).

	R	**W**	**R-W**
Hilo 0	6,1397	40,1578	6,3363
Hilo 1	6,1398	40,1579	6,3363

Tabla 5.13: Resultados RR accesos solapados (segundos).

	R	**W**	**R-W**
Hilo 0	2,0866	18,8695	3,2883
Hilo 1	2,0866	18,8695	3,2883

Tabla 5.14: Resultados LL accesos solapados (segundos).

Este fenómeno puede ser interpretado a través de efectos producidos por el protocolo de coherencia de caché. Según se detalló en la caracterización de la arquitectura QPI (sección 3.1.3), esta permite un intercambio directo de líneas de caché entre nodos. Teniendo esto en cuenta, cuando existe una superposición en los accesos, de acuerdo con la especificación de QPI [1], podríamos esperar cuatro flujos de datos potenciales, determinados por qué hilo realiza el primer acceso a la memoria. Conociendo de antemano que el hilo 0 (local) está ubicado junto con los datos en el nodo 0 y el hilo 1 en el nodo 2 (remoto), estos comportamientos posibles son:

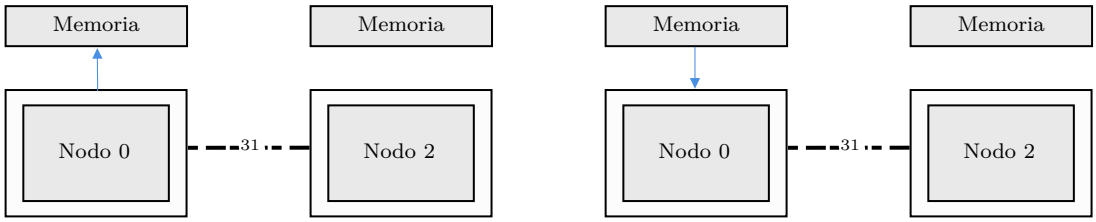
- **Casuística 1:** si el hilo 0 es el primero en acceder a los datos, estos residirán en su memoria local. Así, solicitará a su controlador de memoria la línea de datos correspondiente, como se ilustra en la figura 5.10a. A continuación, el controlador de memoria devolverá la línea de datos solicitada, representado en la figura 5.10b.

- **Casuística 2:** en el caso de que el hilo 1 acceda a los datos en segundo lugar, estos se ubicarán en la caché del nodo 0. Bajo este escenario, el hilo 1 solicitará los datos al nodo 0, tal como se muestra en la figura 5.11a.

Seguidamente, el nodo 0 retornará los datos directamente desde su memoria caché a través del bus QPI, como se ilustra en la figura 5.11b.

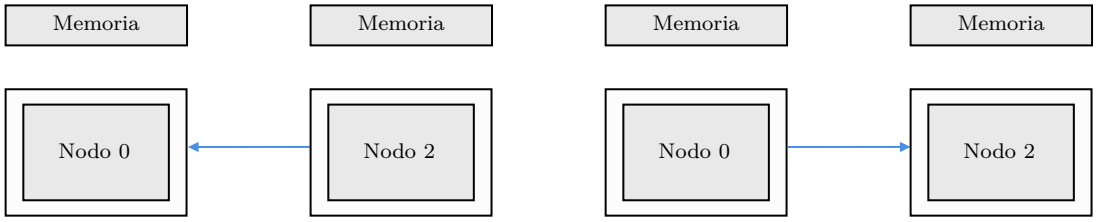
- **Casuística 3:** si el hilo 1 es el primero en acceder a los datos, tendrá que solicitarlos al nodo 0, como se muestra en la figura 5.12a. El nodo 0, al constatar que los datos no se encuentran en su caché, deberá hacer una petición al controlador para obtenerlos de su memoria local, tal como se ilustra en la figura 5.12b. Después de obtener los datos de la memoria, representado en la figura 5.12c, el nodo 0 finalmente enviará los datos al nodo 2 a través del bus QPI, como se puede ver en la figura 5.12d.

- **Casuística 4:** en el escenario donde el hilo 0 es el segundo en acceder a los datos, solicitará la línea correspondiente al nodo 2 (figura 5.13a). A continuación, el nodo 2 remitirá la línea de datos desde su caché utilizando el bus QPI.



(a) Paso 1 (b) Paso 2

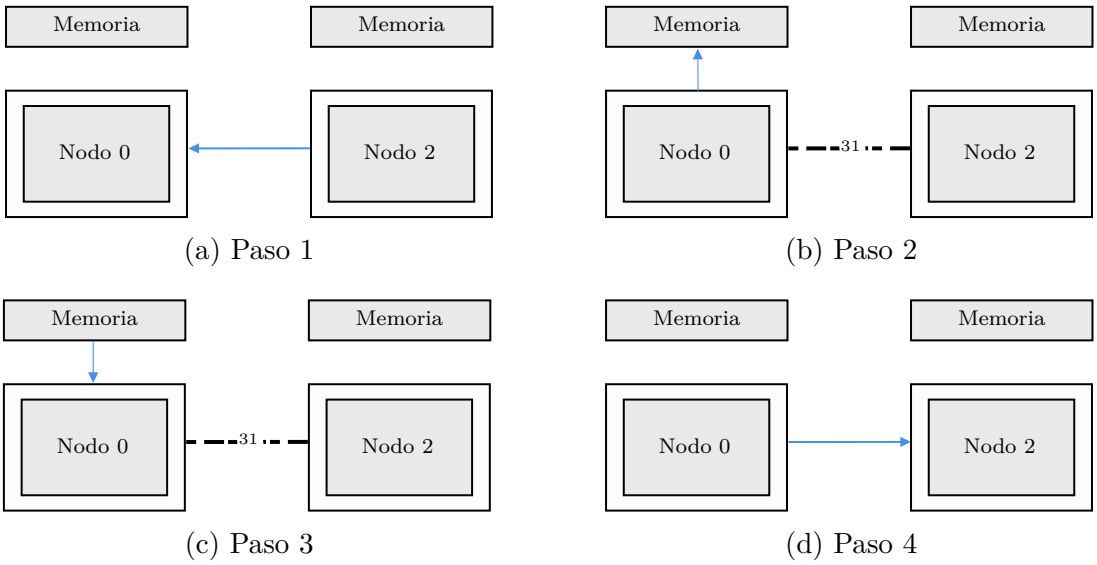
Figura 5.10: Casuística 1: el hilo 0 accede a los datos desde su memoria local.



(a) Paso 1 (b) Paso 2

Figura 5.11: Casuística 2: el hilo 1 accede a los datos desde la caché del nodo 0.

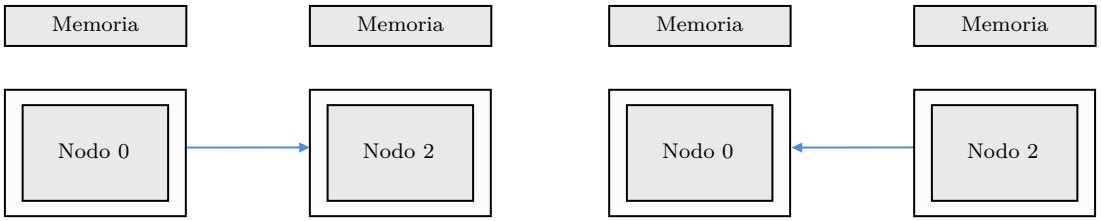
Así, al analizar los comportamientos vinculados a las **casuísticas 1** y **3** vemos cómo, al requerir accesos a memoria, conllevan un mayor coste temporal que las **casuísticas 2** y **4**, donde se accede directamente a los datos desde caché a través de la red NUMA de QPI. Esto resulta en dos secuencias repetitivas de comportamientos en el acceso a los datos:



(a) Paso 1 (b) Paso 2

(c) Paso 3 (d) Paso 4

Figura 5.12: Casuística 3: el hilo 1 accede a los datos desde la memoria del nodo 0.



(a) Paso 1 (b) Paso 2

Figura 5.13: Casuística 4: el hilo 0 accede a los datos desde la caché del nodo 2.

1. Cuando el hilo local accede primero a los datos: en este escenario, la **casuística 1** ocurre en primer lugar, seguida por la **casuística 2**. Dado que la primera es sustancialmente más lenta que la segunda, llegará un punto en el que el orden de los accesos se invertirá y será el hilo local el que acceda primero a los datos.

2. Cuando el hilo remoto accede primero a los datos: en este caso, la **casuística 3** se da primero, seguida por la **casuística 4**. Nuevamente, debido a que la primera es notablemente más lenta que la segunda, habrá un momento en que el hilo remoto sobrepasará al hilo local y será el primero en acceder a los datos.

Así, observamos que estos dos comportamientos se entrelazan y alternan en la realización de los accesos, provocando que los hilos 0 y 1 se vayan adelantando uno al otro hasta que el *array* se haya recorrido por completo.

De este modo, se explica la singular sincronización que se presenta en los

accesos, especialmente notable en la *prueba LR*, reduciendo el rendimiento del hilo local hasta en un 300 %.

5.3.2. Accesos solapados con reducción de trabajo

En este caso, el hilo 0 accede a todos los elementos del vector, mientras que el hilo 1 accede solo a la mitad (ver figura 5.14). En las tablas 5.15 5.16 y 5.17 vemos cómo el tiempo de ejecución del hilo 1 se reduce a aproximadamente la mitad (dado que se están produciendo la mitad de accesos). En el caso del hilo 0, en las *pruebas RR* y *pruebas LL* existe un variación leve del tiempo de ejecución, mientras que en la *prueba LR* el coste se reduce casi hasta la mitad, pese a realizar la misma cantidad de accesos que en la *prueba con accesos solapados*.

Cuando se realizan accesos solapados, debido a la intervención del protocolo de coherencia, el coste de ambos hilos se sincroniza y aumenta levemente. En caso de que los accesos se realicen desde el mismo nodo, esta sincronización no tendrá un efecto demasiado relevante. Sin embargo, cuando el acceso se realiza desde nodos diferente, el coste final se sincroniza al más lento. De esta manera, al reducir el número de accesos del hilo 1 (remoto), aunque el hilo 0 (local) esté realizando el mismo número de operaciones, el coste se reduce a la mitad al evitar su sincronización con el hilo 1 en parte de los accesos al vector.

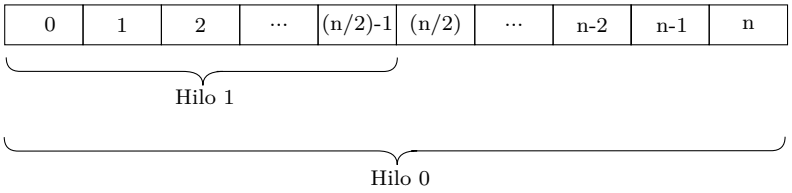


Figura 5.14: Accesos de cada hilo sobre el vector.

	R	W	R-W
Hilo 0	3,1773	20,1871	4,2097
Hilo 1	2,7648	20,0847	3,2438

Tabla 5.15: Resultados LR accesos solapados (segundos).

	R	W	R-W
Hilo 0	6,3795	40,6205	7,1126
Hilo 1	3,3154	20,2910	3,9061

Tabla 5.16: Resultados RR accesos solapados (segundos).

	R	W	R-W
Hilo 0	2,1173	18,5404	3,3843
Hilo 1	1,0916	9,2089	1,7290

Tabla 5.17: Resultados LL accesos solapados (segundos).

5.3.3. Accesos semi-solapados

En este caso, introduciremos un retardo en la ejecución del hilo 1, de forma que al contrario que en la prueba con *accesos solapados con reducción de trabajo*, ambos hilos realizan el mismo número de operaciones pero reduciendo la sincronización en la medida de lo posible.

En este caso, introducimos un retardo de ejecución de 0,5 s al hilo 1 con la función `sleep_for(500ms)` [43].

Para facilitar la comprensión de los datos, estudiaremos los resultados en términos del tiempo de ejecución y el *speedup*.

Este *speedup* se calculará para cada uno de los casos (LL, RR y LR) entre las pruebas con *accesos totalmente solapados* (ver tablas 5.12, 5.13 y 5.14) y las pruebas con accesos semi-solapados. El *speedup* empírico de estos se indicarán con la letra (E) al lado de cada hilo, además se añadirá una fila con el *speedup* hipotético (H) que debería haber sufrido el hilo 1 al incorporar el retardo.

En las tablas 5.18, 5.19 y 5.20 vemos cómo, en el caso de las *pruebas RR* y *LL*, el hilo 1 incrementa su tiempo de ejecución en aproximadamente 0,5 s y el hilo 0 mantiene un coste similar a la ejecución con *accesos solapados*. Sin embargo, en el caso de la *prueba LR* vemos que, en las lecturas, se reduce el tiempo de ejecución tanto del hilo 0 como del hilo 1.

	R	**W**	**R-W**
Hilo 0	4,1190	40,3285	6,0022
Hilo 1	5,9990	41,1233	6,9166

Tabla 5.18: Resultados LR accesos semi-solapados (sleep).

	R	**W**	**R-W**
Hilo 0	6,1972	40,0460	6,3458
Hilo 1	6,7079	40,4668	6,8425

Tabla 5.19: Resultados RR accesos semi-solapados (sleep).

	R	**W**	**R-W**
Hilo 0	2,0669	18,8071	3,3434
Hilo 1	2,5669	19,3013	3,8458

Tabla 5.20: Resultados LL accesos semi-solapados (sleep).

En las tablas de *speedup* (ver tablas 5.21, 5.22 y 5.23) vemos cómo, en el caso de las lecturas, no solo se produce un incremento del 57 % en el rendimiento del hilo 0, sino que también se incrementa el del hilo 1 en, aproximadamente, 0,5 s. Así, vemos como el hilo 1 mejora un 16 % con respecto al retardo hipotético.

Este comportamiento, de nuevo, es achacable al protocolo de coherencia caché. Al introducir un retardo de ejecución en el hilo 1, permitimos que el hilo 0, que se encuentra accediendo a sus datos locales, cargue parte del vector en su caché antes de que el hilo 1 comience su ejecución. De esta manera, cuando el hilo 1 comienza a leer elementos, accede directamente desde la caché del hilo 0 evitando realizar parte de los accesos a memoria, forzando la **casuística 2** (ver figura 5.11).

Además, evitamos que parte de los accesos del hilo 0 se sincronicen con el hilo 1, acercado su tiempo de ejecución al de una prueba con *un hilo y datos privados*. En el caso de las escrituras dado que, estas se realizan en un *buffer* en caché (ver sección 5.1.1), el hilo 1 no se puede aprovechar de la precarga del hilo 0, de forma que se produce una reducción de su rendimiento igual a la hipotética. En el caso de las lecturas-escrituras, vemos cómo el retardo introducido es suficiente para evitar parte de la sincronización con el hilo 0, permitiendo una mejora del rendimiento del 8 %. Sin embargo, en este caso, no permite para esta operación, que el hilo 1 se aproveche de la precarga, resultando en un reducción de su rendimiento cercano al hipotético.

En las *pruebas RR* y *LL*, dado que los accesos se realizan desde el mismo nodo, la sincronización no producía una reducción del rendimiento significativa en ninguno de los hilos. De esta forma, la introducción del retardo no proporciona un beneficio significativo en el hilo 0 e incrementa el tiempo de ejecución del hilo 1 ajustándose al coste hipotético esperado.

	R	**W**	**R-W**
Hilo 0(E)	1,57	1,00	1,08
Hilo 1(E)	1,08	0,98	0,93
Hilo 1(H)	0,92	0,98	0,92

Tabla 5.21: *Speedup* LR accesos semi-solapados (*sleep*).

	R	**W**	**R-W**
Hilo 0(E)	0,99	1,00	0,99
Hilo 1(E)	0,91	0,99	0,92
Hilo 1(H)	0,92	0,98	0,92

Tabla 5.22: *Speedup* RR accesos semi-solapados (*sleep*).

	R	**W**	**R-W**
Hilo 0(E)	1,00	1,00	0,98
Hilo 1(E)	0,81	0,97	0,85
Hilo 1(H)	0,80	0,97	0,86

Tabla 5.23: *Speedup* LL accesos semi-solapados (*sleep*).

En las siguientes secciones se muestra la evolución del rendimiento de los hilos a medida que se aumenta el retardo del hilo 1 entre 0 s y 0,9 s con incrementos de 0,1 s. En concreto, para cada operación de lectura, escritura y lectura-escritura, estudiaremos los efectos de este retado en cada distribución *LR*, *LL* y *RR*.

Lecturas

En las figuras 5.15 , 5.16 y 5.17 y vemos dos comportamientos diferenciados en función de si los hilos realizan o no el acceso a los datos desde el mismo nodo.

1. *Prueba LR*: en este caso, la sincronización produce que el tiempo de ejecución del hilo 0 se incremente de 2,0307 s a 6,4941 s. Así, al aumentar el retardo del hilo 1, que se encuentra en remoto, desincronizamos el acceso de

manera que el hilo 0 puede cada vez acercase más al coste de una ejecución asilada. Por otro lado, el hilo 1 puede aprovecharse de los datos precargados por el hilo 0, mejorando su rendimiento pese al retardo. En la figura 5.15 vemos que el pico de rendimiento del hilo 1 se consigue al introducir un *sleep* de 0,1 s, reduciendo el tiempo de ejecución de 6,5019 s a 5,6727 s. A medida que se incrementa este retardo, el hilo 1 deja de poder aprovecharse de los datos precargados en caché y empieza a reducir su rendimiento hasta volver a un tiempo de ejecución de 6,4925 s al introducir un retardo de 0,9 s.

2. *Prueba LL* y *RR*: en estos caso, la sincronización no afectaba al rendimiento de ninguno de los hilos de forma significativa, dado que ambos tenían el mismo coste de acceso a los datos (al encontrarse los dos en el mismo nodo). Así, el coste con *accesos solapados* ya se acercaba al de una ejecución aislada (ver tablas 5.13 y 5.14). De esta manera, cuando los hilos se sincronizan, se alternan los accesos a memoria, sin obtener ninguna mejora visible. De esta forma, al incrementar el retardo se producen los siguientes comportamientos sobre los hilos:

   - Hilo 0: comienza a tener un comportamiento más errático, aumentando la variabilidad de los resultados cada ejecución. En concreto, en el caso de la *prueba RR* se produce una variación del coste de ejecución de 1 s entre pruebas.
   - Hilo 1: incrementa su tiempo de ejecución a medida que aumentamos el retardo, además, vemos un progresión más estable en la *prueba LL* (ver gráfica 5.16) que en la prueba *RR* (ver gráfica 5.17).

   Ambos comportamientos vuelven a ser explicables por efectos de la caché. Como ambos hilos acceden a los datos desde el mismo nodo, introducir un retardo puede provocar que, en ciertos casos, el hilo atrasado acceda a los datos demasiado tarde y tenga que volver a cargarlos de memoria. En esta situación, ambos hilos comenzarán a competir por la caché incrementándose el número de fallos y afectando al rendimiento. Este hecho se ve agravado al acceder a datos remotos (*prueba RR*), al sumar a esta situación el incremento de los retardos en los accesos a memoria y la competición por los recursos de interconexión de nodos.

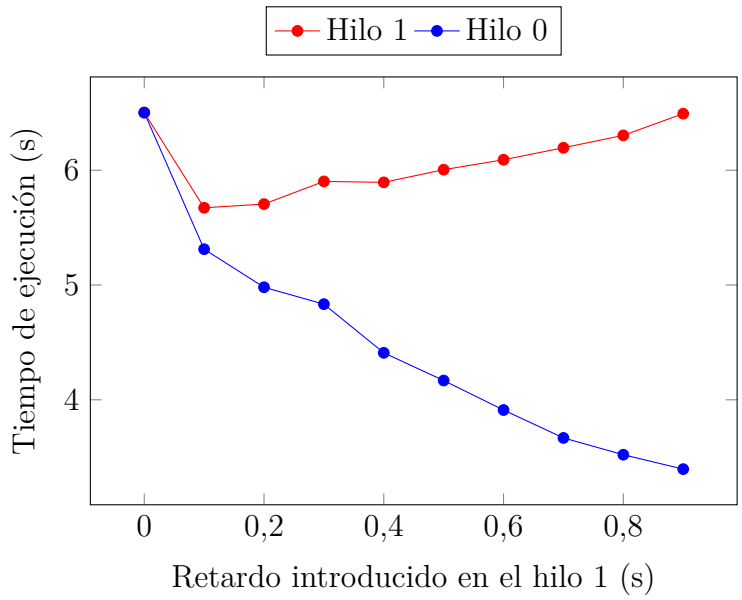


Figura 5.15: Variación del coste de los hilos en función del *sleep* en lecturas (prueba LR).

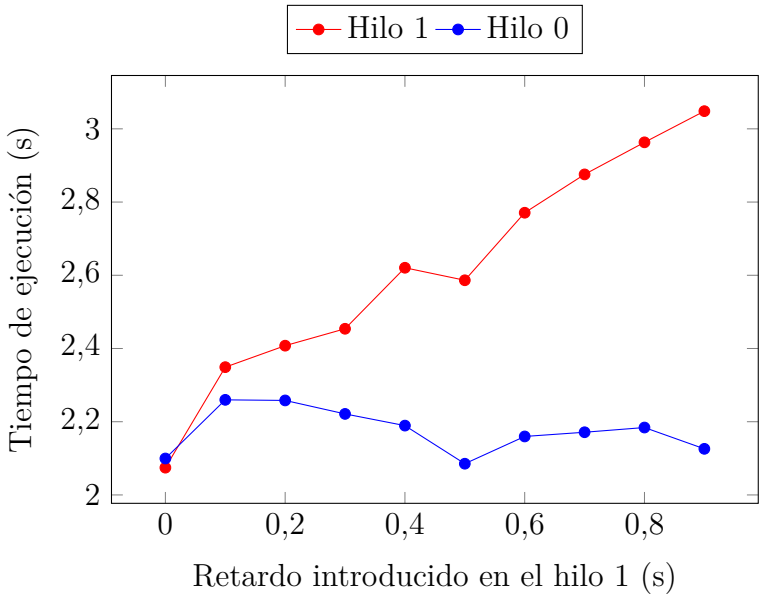


Figura 5.16: Variación del coste de los hilos en función del *sleep* en lecturas (prueba LL).

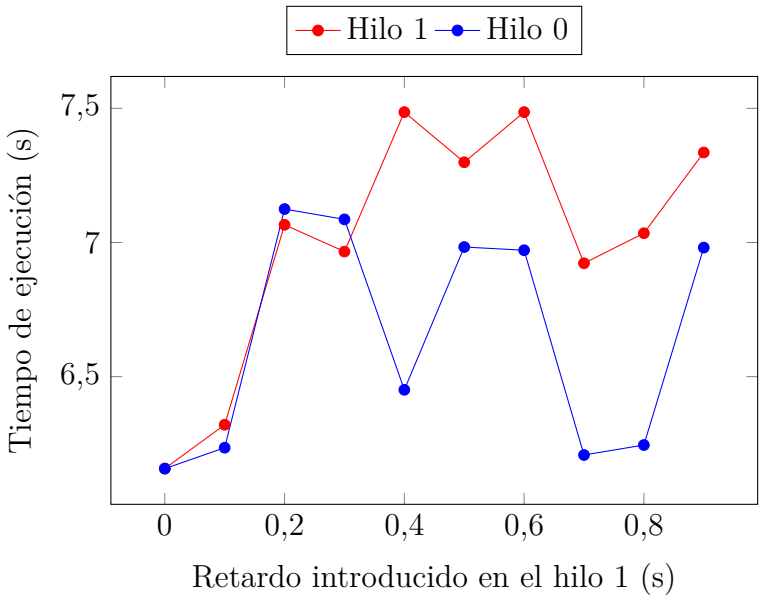


Figura 5.17: Variación del coste de los hilos en función del *sleep* en lecturas (prueba RR).

Escrituras

En la gráfica 5.18, 5.19 y 5.20 vemos de nuevo dos comportamientos diferenciados:

1. *Prueba LR*: en este caso, como ya se mencionó, el hilo 1 no se puede aprovechar de la precarga del hilo 0 (ver sección 5.1.1). De esta manera, por un lado, el hilo 0 mejora su rendimiento a medida que aumenta el retardo del hilo 1, al acercarse más a una ejecución aislada. Por otro lado, el hilo 1 aumenta su coste a medida que se incrementa el retardo sin poder aprovecharse de ningún tipo de precarga.

2. *Prueba RR y LL*: en estos casos, a la situación anterior se suma la competición por los recursos de acceso a memoria y caché, agravándose de nuevo aún más cuando los accesos son a datos remotos en ambos.

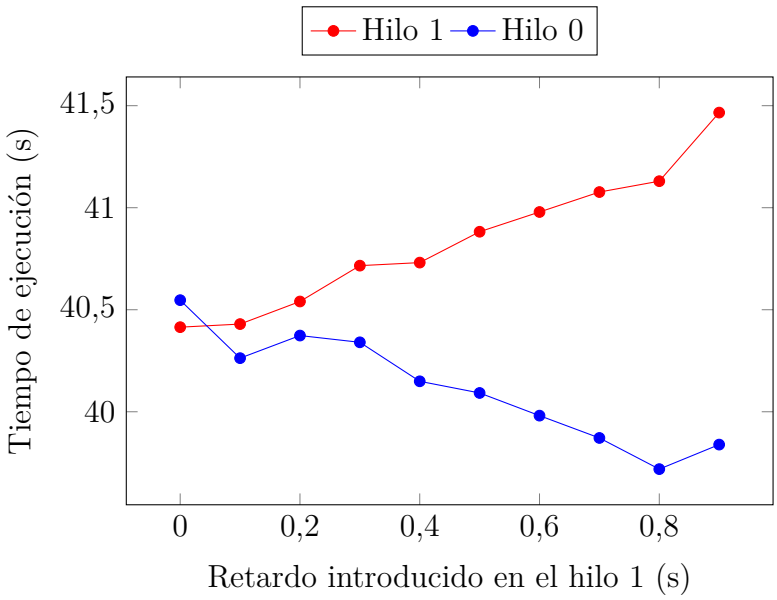


Figura 5.18: Variación del coste de los hilos en función del *sleep* en escrituras (prueba LR).

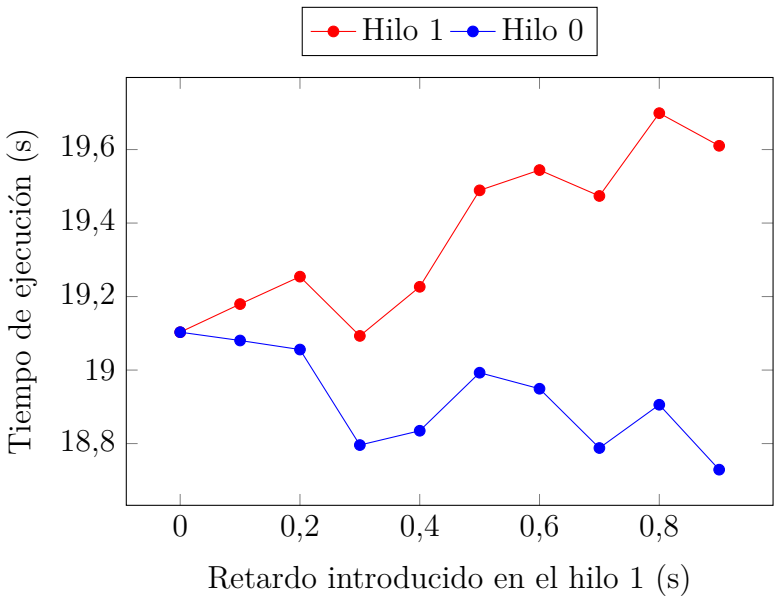


Figura 5.19: Variación del coste de los hilos en función del *sleep* en escrituras (prueba LL).

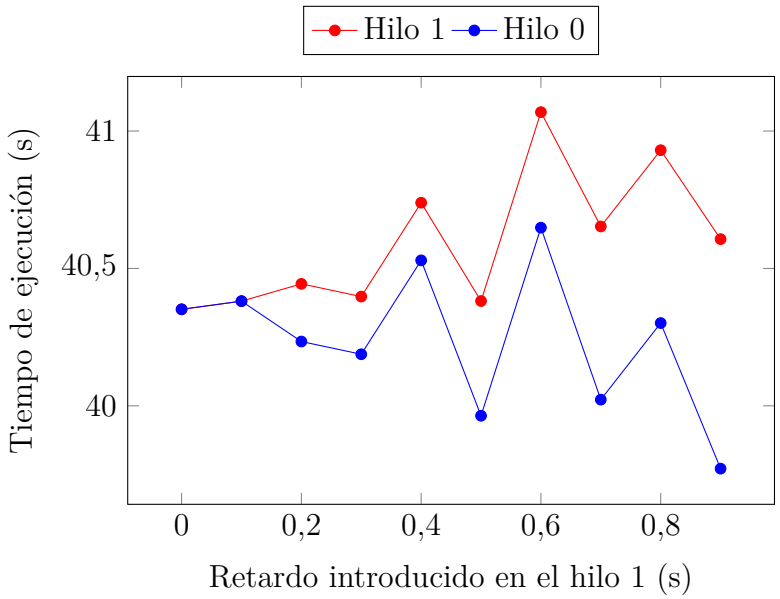


Figura 5.20: Variación del coste de los hilos en función del *sleep* en escrituras (prueba RR).

Lecturas-Escrituras

En las figuras 5.21, 5.22 5.23 vemos cómo volvemos a obtener unos resultados muy similares al de las escrituras. Pese a que el hilo 1 sí accede a datos que el hilo 0 precarga, en la figura 5.21 vemos que su rendimiento no mejora al introducir el retardo. Al realizar modificaciones en dos nodos diferentes interviene el protocolo de coherencia, siguiendo el esquema de la tabla 3.4. Así, además de los datos del vector, se deberán transmitir múltiples mensajes de coherencia, este intercambio extra de información podría estar afectando a posibles mejoras de rendimiento.

En el resto de pruebas expuestas en las gráfica 5.22 y 5.23 vemos cómo obtenemos un comportamiento análogo a los anteriores, con una mayor variación al compartir recursos, en especial cuando los accesos son remotos para ambos hilos.

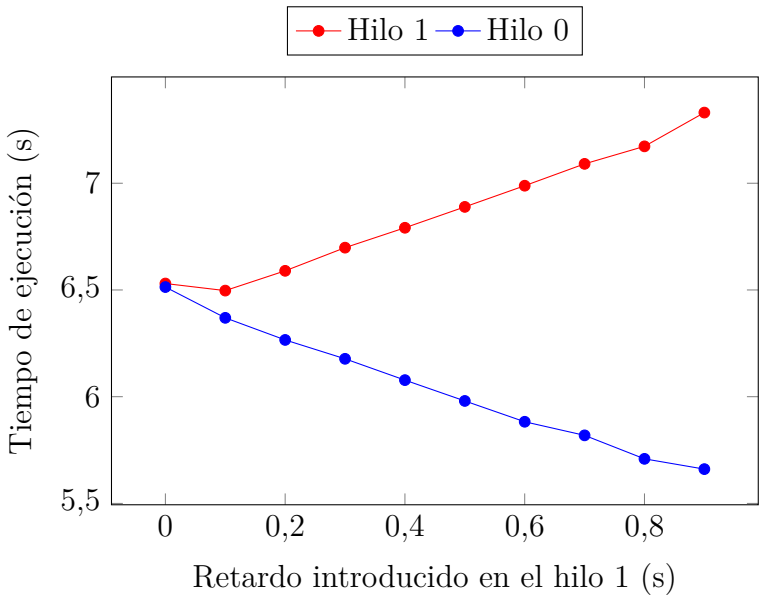


Figura 5.21: Variación del coste de los hilos en función del *sleep* en lecturas-escrituras (prueba LR).

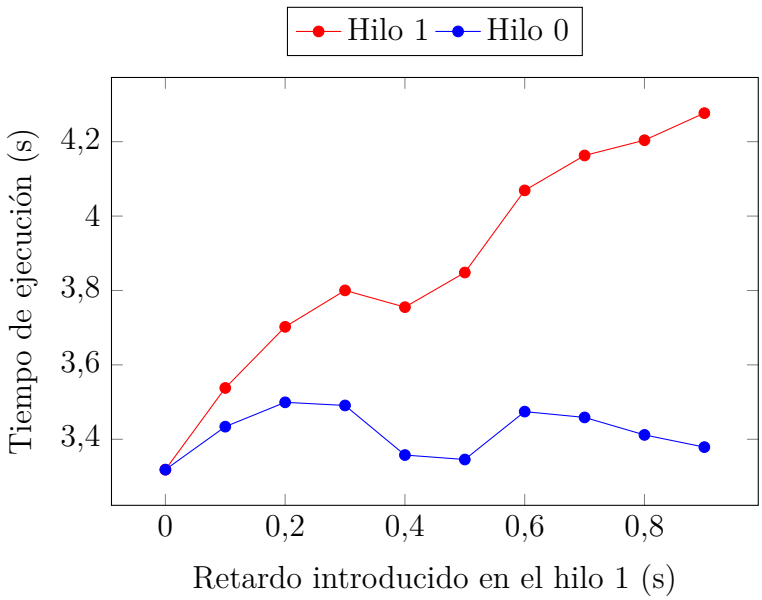


Figura 5.22: Variación del coste de los hilos en función del *sleep* en lecturas-escrituras (prueba LL).

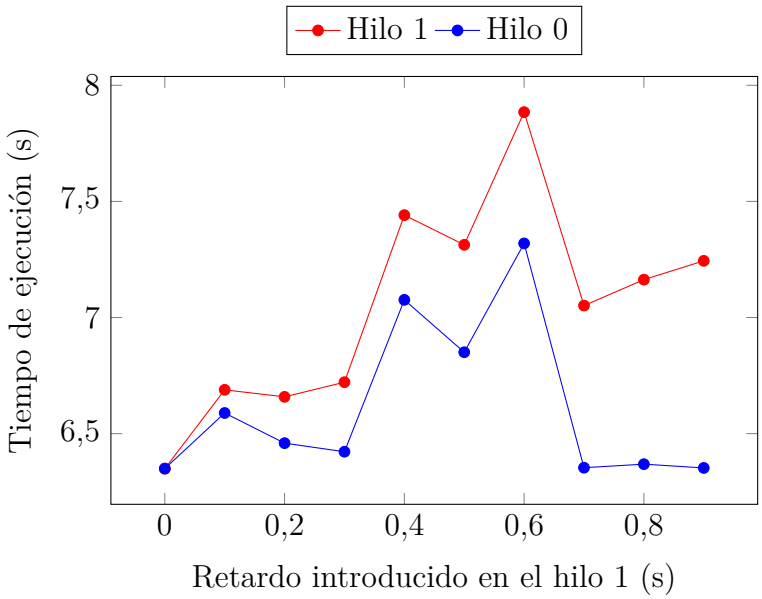


Figura 5.23: Variación del coste de los hilos en función del *sleep* en lecturas-escrituras (prueba RR).

5.3.4. Recapitulación

La distribución más afín es aquella en la que ambos hilos son locales a los datos. Además se observó cómo, primero, el protocolo de coherencia caché sincroniza a los hilos cuando los accesos están solapados, este hecho es especialmente relevante cuando los hilos se encuentran en nodos distintos, sincronizándose al más lento. Segundo, cuando se reduce el número de accesos de los hilos al vector, el otro comienza a acercarse al comportamiento de una ejecución aislada. Por último, si introducimos un retardo en el acceso a los datos en uno de los hilos, podemos conseguir, a medida que aumenta, que el hilo sin el retardo comience a acercarse al comportamiento que tendría en una ejecución aislada. Este efecto se ve muy influenciado por la distribución de los hilos y las operaciones realizadas, si existe un hilo local a los datos y otro remoto (*Prueba LR*), introduciendo el retardo adecuado, el último se puede aprovechar de los datos que el hilo local vaya cargando en su caché, mejorando el tiempo de ejecución del hilo remoto pese al retardo. Esta situación se da solo en las lecturas, en el caso de las escrituras y lecturas-escrituras, no existen mejoras en el hilo remoto por problemas con el protocolo de coherencia y la estructura de la jerarquía de memoria. En las operaciones con dos hilos locales o dos hilos remotos pertenecientes al mismo nodo (*pruebas RR* y *LL*), debido a que la sincronización no producía un impacto negativo relevante y a la compartición de la caché y los buses de interconexión, los comportamientos comienzan a ser más erráticos con variaciones más abruptas en los tiempos de ejecución de los hilos para un mismo índice de retardo.

Capítulo 6

Conclusiones y posibles ampliaciones

Una vez obtenidos y estudiados los resultados, podemos resumir las principales aportaciones del trabajo, así como exponer futuras ampliaciones que permitirían caracterizar por completo las arquitecturas NUMA.

6.1. Principales aportaciones

En este trabajo, hemos analizado la influencia de varios factores en el rendimiento de un sistema NUMA conformado por cuatro procesadores Intel Xeon E5-4620 v4 interconectados en anillo mediante el bus Intel QPI. El objetivo principal ha sido identificar la configuración más eficiente en términos de distribución de hilos y datos sobre los nodos, así como de compartición del procesador y solapamiento en el acceso a datos compartidos.

Utilizando programas de pruebas intensivos en accesos a memoria, hemos analizado distintas configuraciones y cuantificado su eficiencia en términos del tiempo de ejecución. Siguiendo este enfoque, las pruebas se diseñaron con el objetivo de obtener conclusiones claras y aplicables, que no estén oscurecidas por efectos de la precarga, la localidad espacial o diferentes optimizaciones del compilador, facilitando la reproducibilidad y ampliación de los experimentos.

Hemos constatado que, en primer lugar, para una única pareja hilo-nodo, la distribución más eficiente es aquella en la que el hilo y los datos se ubican en el mismo nodo, garantizando accesos a memoria locales. Además, debido a los mecanismos de escritura de datos por los que se hace un *bypass* de la caché, el coste de realizar escrituras aisladas es más de 6 veces superior al de realizar operaciones de lectura o de lectura-escritura.

En segundo lugar, cuando dos hilos paralelos acceden a su propio conjunto de datos privado, la distribución más afín vuelve a ser aquella donde los datos son locales. No obstante, existe un efecto asociado a la compartición de recursos de

los hilos durante la ejecución. En el caso de las operaciones de lectura y lectura-escritura, si estudiamos la compartición individualmente el mayor efecto viene cuando se comparte el bus QPI, con un *overhead* de casi el 14 % en el peor de los casos. El peor escenario se da cuando se comparten todos los recursos estudiados, aumentando el coste de la ejecución hasta un 14,75 %. En el caso de la escrituras, al producirse un *bypass* de la caché, el factor dominante es la latencia de memoria, quedando enmascarados los posibles efectos de la compartición de recursos.

Por último, cuando ambos hilos acceden al mismo conjunto de datos, la distribución más afín es aquella en la que los accesos a los datos son locales. Además, se observó que, debido al funcionamiento del protocolo de coherencia caché, se produce una sincronización entre hilos. Esta situación es especialmente relevante cuando uno de los hilos realiza los accesos localmente y otro en remoto, dado que ambos se sincronizarán al más lento. Este efecto se ve muy influenciado por el momento de acceso a los datos, en caso de introducir un retardo en el hilo remoto, permitimos que el local precargue en su caché parte de los datos a los que accederá, reduciendo el coste del hilo remoto y mejorando el rendimiento del local al acercarse al tiempo de una ejecución aislada. Sin embargo, este efecto varía en función de la operación y la distribución de los hilos y los datos, resultado en ejecuciones con mucha variabilidad en caso de que los hilos compartan nodo y, en ejecuciones constantes pero sin mejora de rendimiento del hilo remoto, en caso de no compartir nodo y realizar operaciones de escritura o lectura-escritura.

Es importante destacar que estas observaciones son específicas para la arquitectura y el protocolo de coherencia de caché utilizados en este estudio, y podrían variar con otras configuraciones. A pesar de ello, los hallazgos aquí presentados proporcionan un valioso punto de partida para futuras investigaciones sobre la optimización del rendimiento en sistemas NUMA.

6.2. Posible trabajo futuro

Este trabajo se centra en la caracterización de comportamientos básicos asociados con uno o dos hilos y un conjunto de datos privado o compartido. El siguiente paso lógico es la generalización de estos comportamientos para la posible obtención de un modelo que permita obtener predicciones sobre el rendimiento de las distintas distribuciones. En concreto, para ello, sería necesario aumentar el conjunto de pruebas de manera que cubran los siguientes aspectos:

- Generalizar las pruebas realizadas sobre los efectos de la compartición de recursos, a medida que incrementa el número de hilos paralelos con datos privados en ejecución. Específicamente, sería de interés determinar, para cada recurso, cómo crece el *overhead* a medida que aumenta el número de hilos paralelos que lo comparten. De esta manera, si el *overhead* pudiese ser modelizado, podría emplearse dicho modelo para predecir posibles efectos de la compartición sobre el rendimiento.

- Estudiar el efecto de la compartición de datos cuando se incrementa el número de hilos. De manera que, se pueda determinar si, además de la sincronización, existen otros efectos sobre el rendimiento asociados a intensificar el número de accesos.
- Ampliar el abanico de operaciones sobre el que se realizan las pruebas, además de realizar mediciones sobre combinaciones de operaciones como lectura-escritura-lectura, escritura-lectura-escritura etc.
- Determinar el efecto de la variación del tamaño del conjunto de datos, desde conjuntos que ocupen los distintos niveles de caché, hasta conjuntos que llenen gran parte de la memoria principal.
- Estudiar el efecto de realizar accesos a múltiples conjuntos de datos por un mismo hilo, viendo posibles comportamientos asociados a la multiplexación en el acceso a diferentes grupos de datos.
- Realizar un estudio en mayor profundidad sobre los efectos de modificar el grado de solapamiento, variando el índice de compartición de los datos. Pudiéndose, por ejemplo, en función del hilo, realizar accesos en orden distinto o realizar múltiples accesos a determinandos datos del vector.

Todo ello, permitiría caracterizar comportamientos más complejos de los sistemas NUMA. De esta manera, se podría obtener parte de la información necesaria para desarrollar un modelo que permitiese realizar predicciones sobre el rendimiento de las distribuciones de hilos y datos, permitiendo así emplear siempre la configuración más eficiente en cada ejecución.

Apéndice A

Manual del *benchmark*

El `benchmark` está compuesto por tres archivos principales, `affinity.cpp`, `parser.hpp` y `types.h` y dos archivos de apoyo, `numaalloc.hpp` y `CLI11.hpp`. Los primeros 3 archivos son los más susceptibles a modificaciones para la creación de nuevas pruebas, en cambio los otros dos ofrecen funciones auxiliares útiles (como `numa_alloc_onnode`) para reservar memoria en un nodo específico.

A.1. Características generales

Los 3 archivos principales poseen las siguientes funciones:

- `affinity.cpp`: contiene toda la lógica de las operaciones sobre las que se realizarán las pruebas, incluyendo:
  - Funciones necesarias para realizar la medición de tiempo.
  - Gestión para la creación de hilos con OpenMP.
  - El código de las funciones que se probarán.
  - El código necesario para imprimir los resultados de forma visual.
  - El código del cálculo del *speedup* de las operaciones elegidas.
- `parser.hpp`: contiene el código para procesar el archivo con los argumentos de la ejecución.
- `types.h`: contiene los tipos de datos que se emplearán durante la ejecución. Por defecto, contiene el tipo de datos del vector y el tipo de datos asociado a su tamaño.

A.2. Modificaciones sobre el software

El código se diseñó de forma que fuese fácilmente adaptable a los requerimiento específicos de las pruebas que quieran realizarse. En concreto, por cada archivo, la estructura es la expuesta a continuación.

A.2.1. `Affinity.cpp`

Como ya se expuso, en él se introducirán las pruebas a realizar y permitirá la obtención de costes de ejecución y *speedup* entre los distintos experimentos.

En específico, dispone de una clase llamada `Thread_array` que contendrá la información privada de cada hilo. Dentro de esta clase, se definirán las pruebas a realizar como una función nueva. Todas la funciones definidas deberán tener las estructura definida en el *template* del código A.1 dónde, se recibe como argumento el tamaño del vector en un tipo de dato `vectorSize` y se devuelve un `int`.

```
1 typedef int (Thread_array::*MemFuncPtr)(vectorSize);
```

Código A.1: *Template* de todas las funciones.

Una vez creada la función para la realización de las pruebas en la clase, es necesario modificar el contenido de la función `setFunctionsPointers` para añadirla al mapa de funciones disponibles y asignarle un identificador. En el código A.2 vemos un ejemplo de, cómo añadir la función "ejemplo" y definir el identificador "ej".

```
1 unordered_map<std::string, MemFuncPtr> setFunctionsPointers(){
2     functMap["ej"] = &Thread_array::ejemplo;
3     return functMap;
4 }
```

Código A.2: *Template* de todas las funciones.

Para poder emplear otros argumentos además del vector a recorrer, es posible acceder dentro de cada función al *hashmap* "arg", que contiene todos los argumentos definidos en el archivo procesado por el *parser* (del cual hablaremos a continuación). En el código A.3 vemos un ejemplo de una función que, además de emplear el tamaño del vector, emplea un argumento llamado *sleep* obtenido del *hashmap*.

```
int read_s(vectorSize size){
    vectorSize i = 0;
    volatile int add = 0;

    int sleep = stoi(arg["sleep"]);
    if(omp_get_thread_num() == 1){
        this_thread::sleep_for(chrono::milliseconds(sleep));
    }
    for(i = 0; i < size; i++){
        add = priv_shared_vec[i];
        i+=dist;
    }
    return add;
}
```

Código A.3: *Template* de todas las funciones.

Por último, todos los hilos acceden al vector de datos empleado el puntero `priv_shared_vec` dado que este es local al hilo. Además, es necesario destacar que:

- El mapa de argumentos es local y privado para cada hilo para evitar problemas de sincronización al realizar los accesos. La implementación interna del mapa es un *hashmap* por lo que el acceso es $\mathcal{O}(1)$ para todos los argumentos, evitando variaciones artificiales sobre las ejecuciones.
- Todos los argumentos definidos en el mapa son `strings` por lo que, es necesario convertirlo al tipo de dato requerido antes de emplearlos.

De esta manera, para la incorporación de nuevas pruebas, únicamente es necesario añadir el código como una función nueva en la clase `Thread_array` e incorporarla al *hashmap* indicando el identificador.

A.2.2. `Parser.hpp`

Este archivo procesa y almacena los argumentos contenidos en otro archivo externo de nombre a elegir. Por defecto, permite procesar un archivo con la estructura expuesta en el código A.4: En este archivo se especifica:

- `opType`: indica las pruebas a realizar separadas por un espacio. Las operaciones se indicarán introduciendo el identificador definido en el archivo `affinity.cpp`.
- `nodePerThread`: indica el nodo donde se localizará cada hilo separado por espacios. Vemos como el número de hilos es intrínseco a este argumento, si por ejemplo se indica `0 1`, se crearán dos hilos, en concreto, el hilo 0 se creará en el nodo 0 y el hilo 1 se creará en el nodo 1.

- `nodePerVector`: indica el nodo donde se localizarán los vectores de datos. De nuevo, el número de hilos que se creará dependerá del número de nodos especificados. En el código A.4, se crearán dos conjuntos de datos en el nodo 0.

- `tamPerVector`: define el tamaño de los vectores creados. El número de tamaños deberá coincidir con el número de vectores especificados en el argumento `nodePerVector`.

- `vectorPerThread`: define qué hilos accederán a cada vector. En concreto, se separarán por comas los identificadores de los hilos que acceden a cada vector. Así en el código A.4 vemos como al vector 0 accederán los hilos 0 y 1, y al vector 1 accederán los hilo 2 y 3.

- `summaryType`: indica el tipo de resumen que se mostrará en el archivo de salida, pudiendo ser de tipo "max" para mostrar el tiempo de ejecución mayor de entre todos los hilos, "min" para mostrar el menor o "sum" para mostrar la suma de todos los tiempos de ejecución.

- `speedup`: indica qué *speedups* se calcularán entre el conjunto de pruebas realizadas. En el código A.4 vemos como se calculará la relación entre el coste de las escrituras y las lecturas. Es posible calcular tantos *speedups* como se crean necesarios separándolos por espacios.

- `numIter`: indica el número de veces que se repetirán cada una de las pruebas. El resultado mostrado por hilo será el cálculo de la media de todas estas repeticiones.

- `argsExtra`: se emplea para introducir tantos argumentos extra como se consideren oportunos, separándolos por espacios en blanco. Por cada argumento se indica un identificador (cualquier cadena de caracteres sin espacios), seguido de dos puntos y el valor del argumento.

```
1 opType: write read
2 nodePerThread: 0 1 2 3
3 nodePerVector: 0 0
4 tamPerVector: 209715200 209715200
5 vectorPerThread: 0 1, 2 3
6 summaryType: max
7 speedupCalc: write/read
8 numIter: 10
9 argsExtra: sleep:100 stride:16
```

Código A.4: Archivo de argumentos.

Los argumentos deben seguir siempre este orden, en caso que desear modificar, eliminar o añadir argumentos será necesario realizar ajustes sobre la función `parse_input_file`.

A.2.3. `types.h`

En este archivo se introducirán los tipos de datos que se emplearán en la ejecución. Por defecto, el contenido del archivo es el mostrado en el código A.5, en este caso, se indica que el tipo de datos del vector sobre el que se realizarán los accesos será `char` y el tipo de dato del tamaño del vector será un `long long int`.

```
1 #ifndef MYTYPES_H
2 #define MYTYPES_H
3
4 typedef char vectorType;
5 typedef long long int vectorSize;
6
7 #endif // MYTYPES_H
```

Código A.5: Archivo de definicón de tipos de datos.

A.3. Recomendaciones al crear pruebas

Para que las pruebas ofrezcan resultados consistentes, es importante tener en consideración:

1. Las variables relevantes empleadas deberían ser privadas y no estáticas dentro de la clase `Tread_array`. De esta forma, garantizamos que se localicen en el nodo del hilo asociado a la instancia del objeto creada.

2. Aunque todo el código es susceptible a adaptaciones según se considere necesario, en general se recomienda.

   - Emplear el cuerpo de función estándar indicado.
   - En caso de necesitar más argumentos emplear el *hashmap*.

3. Para evitar optimizaciones del compilador, es recomendable:

   - Que exista una variable que se emplee cada vez que se realizan la/s operaciones a medir, además de devolver el valor de la variable al final de la función. De esta manera, la variable estará sujeta a la función `doNotOptimizeAway` y no se realizarán optimizaciones ni reordenaciones con ella.

- Emplear el *qualifier* `volatile` en los iteradores de los bucles, para evitar que el compilador los elimine.
- Obtener el ensamblador del código antes de realizar la ejecución, asegurándonos de que las operaciones no están siendo ignoradas u optimizadas agresivamente por el compilador

4. Emplear la función `getAdressNode` para confirmar que las reservas de memoria se están realizando correctamente en los nodos deseados.

A.4. Ejecuciones de pruebas

Para la ejecución de pruebas, es necesario:

1. Definir los tipos de datos a emplear en el archivo `types.h` y crear la función de pruebas en el archivo `affinity.cpp`, tal y como se indicó en las secciones A.2.1 y A.2.3.

2. Definir el archivo de argumentos deseado siguiendo el esquema de la sección A.2.2.

3. Compilar el código con un compilador de C++, y las liberías `-lnuma` y `-fopenmp`.

4. Ejecutar el código, indicado en primer lugar el nombre del archivo de argumentos y, en segundo lugar, el nombre del archivo de salida. Un ejemplo sería, `./affinity.out args salida`.

Tras cada ejecución se obtendrá una salida como la expuesta en el código A.6, donde se muestra en la parte superior la información de la ejecución y en la parte inferior los resultados obtenidos por cada hilo para cada operación.

```
                    --Execution data--
+ Vectors data
 - Vector 0     node: 0     size: 209715200


+ Threads data
 - Thread 0     node: 0     vector: 0
 - Thread 1     node: 2     vector: 0

+ Operation data
 - Operations to be performed: read_s, write_s, read_write_s,
 - type of summary: max

                         -Results-
                                 Time_Cost(s)
               | Thread 0:      6.501979872
read_s         | Thread 1:      6.501987319
               | Total:         6.501987319

               | Thread 0:      40.546937634
write_s        | Thread 1:      40.614545821
               | Total:         40.614545821

               | Thread 0:      6.513637034
read_write_s   | Thread 1:      6.530190013
               | Total:         6.530190013
```

Código A.6: Archivo de salida de ejecución.

Apéndice B

Reproducibilidad de experimentos

Para la ejecución de todas las pruebas expuestas en el capítulo 4, es necesario, una vez descargado el código, ejecutar el *script* `pruebas/ejec.sh`. Este recorrerá las carpetas que contengan la palabra "pruebas" al principio, tal y como se muestra en el código B.1.

```
#!/bin/bash
archivo="../affinity"
g++ "$archivo.cpp" -lnuma -fopenmp -O0 -o "$archivo.out"

#Ejecutamos todas las pruebas
find . -type f -name "salida*" -exec rm -f {} \;
carpetas=$(find . -maxdepth 1 -type d -name "pruebas*")
IFS=$'\n' read -rd '' -a carpetas <<< "$carpetas"

for elemento in "${carpetas[@]}"; do
    cd "$elemento"
    carpetasAux=$(find . -maxdepth 1 -type d -name "pruebas*")
    IFS=$'\n' read -rd '' -a carpetasAux <<< "$carpetasAux"

    if [ -z "$carpetasAux" ]; then
        sh ejec.sh
    else
        for elementoAux in "${carpetasAux[@]}"; do
            cd "$elementoAux"
            sh ejec.sh
            cd ..
        done
    fi
    cd ..
done
```

Código B.1: Archivo de salida de ejecución.

En concreto existen 5 carpetas con las pruebas ya explicadas:

- `pruebas_un_hilo`: con las pruebas de *un hilo y datos privados*.
- `pruebas_solo_priv`: con las pruebas de *dos hilos y datos privados*.
- `pruebas_solape`: con las pruebas de *accesos solapados*.
- `pruebas_half_access`: con las pruebas de *accesos solapados con reducción de trabajo*.
- `pruebas_sleep`: con las pruebas de *accesos semi-solapados*.

Una vez termine la ejecución, en cada carpeta aparecerá un archivo con el nombre "salidaVisual" con los resultados de cada prueba, siguiendo la estructura especificada en la sección A.4.

Bibliografía

[1] "An Introduction to the Intel QuickPath Interconnect," Document Number: 320412-001US, 2009. [En línea]. Disponible en: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[2] D. Padua, "Encyclopedia of Parallel Computing," en *Springer*, New York, 2011. ISBN: 978-0-387-09765-7.

[3] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?" en *Computer*, vol. 30, no. 9, pp. 37-39, Sept. 1997.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," en *USENIX SEDMS*, 2011.

[5] D. Zeng, L. Zhu, X. Liao y H. Jin, "A Data-Centric Tool to Improve the Performance of Multithreaded Program on NUMA," en *Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015. Proceedings, Part IV*, vol. 9531, pp. 74-87, 2015. [En línea]. Disponible en: https://doi.org/10.1007/978-3-319-27140-8_6

[6] N. Manchanda and K. Anand, "Non-Uniform Memory Access (NUMA)," *New York University*, May 2004.

[7] R. P. LaRowe, Jr., "Page placement for nonuniform memory access time (NUMA) shared memory multiprocessors," ProQuest Dissertations and Theses, pp. 292, 1991. [En línea]. Disponible en: https://www.proquest.com/dissertations-theses/page-placement-nonuniform-memory-access-time-numa/docview/303948220/se-2

[8] I. Sánchez Barrera, "Exploiting data locality in cache-coherent NUMA systems," Tesis Doctoral, UPC, Departamento de Arquitectura de Computadores, 2022. [En línea]. Disponible en: http://hdl.handle.net/2117/367546

[9] Y. Wang, D. Jiang y J. Xiong, "NUMA-aware thread migration for high performance NVMM file systems," en *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*, 2020.

[10] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra y P. Beckman, "Exploring data migration for future deep-memory many-core systems," en *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 289–297, 2016.

[11] "NAS Parallel Benchmarks," NASA Advanced Supercomputing (NAS) Division, [En línea]. Disponible en: `https://www.nas.nasa.gov/software/npb.html`. Accedido el: 10 de mayo, 2023.

[12] "SPEC OMP 2012," Standard Performance Evaluation Corporation, [En línea]. Disponible en: `http://www.spec.org/omp2012/`. Accedido el: 10 de mayo, 2023.

[13] "STREAM," Department of Computer Science, [En línea]. Disponible en: `https://www.cs.virginia.edu/stream/ref.html`. Accedido el: 10 de mayo, 2023.

[14] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, y P. Cao, "How to Build a Benchmark," en *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*, Austin, Texas, USA, pp. 333–336, 2015. [En línea]. Disponible en: `https://doi.org/10.1145/2668930.2688819`

[15] M. Farrens, G. Tyson, and A. R. Pleszkun, "A Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors," en *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 338–347, Apr. 1994.

[16] M. Farrens, G. Tyson, and A. R. Pleszkun, "A Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors," en *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 338–347, IEEE Computer Society Press, Washington, DC, USA, 1994.

[17] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-Aware Scheduling on Multicore Systems," en *ACM Trans. Comput. Syst.*, vol. 28, no. 4, Art. no. 8, Dec. 2010.

[18] Standard C++ Foundation, "Current Status," 2023. [En línea]. Disponible en: `https://isocpp.org/std/status`

[19] A. Kleen, "A NUMA API for Linux," *Novel Inc*, 2005.

[20] R. J. Safranek, D. D. Sharma, and G. N. Srinivasa, "Implementing QuickPath Interconnect Protocol Over a PCIe Interface," United States Patent Application Publication No. US 2012/0079156 A1, Mar. 29, 2012. [En línea]. Disponible en: https://patentimages.storage.googleapis.com/22/fe/6d/b38e3415ca8f26/US20120079156A1.pdf

[21] A. W. Hay, "MESIF Cache Coherency Protocol," Master's Thesis, The University of Auckland, 2012. [En línea]. Disponible en: https://www.scss.tcd.ie/~jones/vivio/caches/Andrew%20Hay%20MESIF%20Cache%20Coherency%20Protocol%202012.pdf

[22] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures," en *2010 18th IEEE Symposium on High Performance Interconnects*, Intel Corporation, Santa Clara, USA, 2010. [En línea]. Disponible en: https://www.cl.cam.ac.uk/~djm202/pdf/papers/ziakas10_intel_qpi_features.pdf

[23] D. L. Mulnix, "Intel Xeon Processor Scalable Family Technical Overview," ID 673025, Updated Dec. 1, 2022. [En línea]. Disponible en: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html

[24] B. Lepers, A. Fedorova, and V. Quema, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," Usenix, [En línea]. Disponible en: https://www.usenix.org/system/files/conference/atc15/atc15-paper-lepers.pdf. Accedido el: 20 de mayo, 2023.

[25] D. Goodman, R. Haecki, and T. Harris, "Modeling memory bandwidth patterns on NUMA machines with performance counters," arXiv, [En línea]. Disponible en: https://arxiv.org/pdf/2106.08026.pdf. Accedido el: 21 de mayo, 2023.

[26] D. Goodman, G. Varisteas y T. Harris, "Pandia: Comprehensive Contention-sensitive Thread Placement," en *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, ACM, pp. 254-269, 2017. [En línea]. Disponible en: https://doi.org/10.1145/3064176.3064177

[27] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, y M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," SIGPLAN Not., vol. 48, no. 4, pp. 381–394, Mar. 2013. Disponible en: https://doi.org/10.1145/2499368.2451157.

[28] GCC Command Options, "3.11 Options That Control Optimization." Disponible en: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-O0.

[29] Documentación de cppreference, "cv (const and volatile) type qualifiers." 2023. Disponible en: https://en.cppreference.com/w/cpp/language/cv.

[30] Prevent GCC from optimizing away a snippet of code, 27 Abr 2015. Disponible en: https://programfan.github.io/blog/2015/04/27/prevent-gcc-optimize-away-code/.

[31] R. Knauerhase, P. Brett, B. Hohlt, T. Li y S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," en *IEEE Micro*, vol. 28, no. 3, pp. 54-66, 2008. [En línea]. Disponible en: https://doi.org/10.1109/MM.2008.48

[32] D. Lemire, "Is reading memory faster than writing on a PC?", Daniel Lemire's blog. Disponible en: https://lemire.me/blog/2012/10/29/is-reading-memory-faster-than-writing-to-memory-on-a-pc/.

[33] "What is Write Amplification?", The SSD Guy blog. Disponible en: https://thessdguy.com/what-is-write-amplification/.

[34] "Cache Write Policies", Sonoma State University. Disponible en: https://rivoire.cs.sonoma.edu/cs351/other/cache_write.html.

[35] U. Drepper, "What Every Programmer Should Know About Memory", Red Hat, Inc., Nov. 21, 2007. Disponible en: https://akkadia.org/drepper/cpumemory.pdf

[36] "Intel VTune Profiler Documentation", Intel Corporation. Disponible en: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-download.html

[37] H.-Y. Cheng, M. J. Irwin, y Y. Xie, "Adaptive Burst-Writes (ABW): Memory Requests Scheduling to Reduce Write-Induced Interference," Pennsylvania State University y University of California, Santa Barbara. [En línea]. Disponible en: https://dl.acm.org/doi/10.1145/2753757.

[38] "External Memory Interface Handbook Volume 2: Design Guidelines: For UniPHY-based Device Families," 9.4.1.10. Bus Turnaround Time. [En línea]. Disponible en: https://www.intel.com/content/www/us/en/docs/programmable/683385/17-0/bus-turnaround-time.html

[39] Y.-S. Moon, Y. Kwon, H.-S. Kim, D.-G. Kim, H. H. Lee, and K. Park, "The Compact Memory Scheduling Maximizing Row Buffer Locality," SK Hynix Inc. [En línea]. Disponible en: https://users.cs.utah.edu/~rajeev/jwac12/papers/paper_3.pdf

[40] "Write Combining," Mechanical Sympathy, 15-Jul-2011. [En línea]. Disponible en: https://mechanical-sympathy.blogspot.com/2011/07/write-combining.html?m=1

[41] "Intel 64 and IA-32 Architectures Optimization Reference Manual," Order Number: 248966-026, Intel, Apr. 2012. [En línea]. Disponible en: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[42] "Write Combining Memory Implementation Guidelines," Order Number: 244422-001, Intel, Nov. 1998. [En línea]. Disponible en: https://download.intel.com/design/PentiumII/applnots/24442201.pdf

[43] "std::this_thread::sleep_for," cppreference.com. [En línea]. Disponible en: https://en.cppreference.com/w/cpp/thread/sleep_for