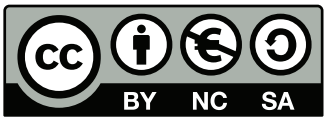
Arquitectura de Computadores

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM



Arquitectura de Computadores

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1. Tema 1

1.1. Lección 1. Clasificación del paralelismo implícito en una aplicación.

1.1.1. Clasificación del paralelismo

Según su naturaleza, podemos clasificar el paralelismo implícito en tres categorías.

Nivel de paralelismo implícito El paralelismo puede darse a diferentes niveles: a nivel de programa, a nivel de funciones, a nivel de bloques, a nivel de operaciones, etc.

Granularidad Es el volumen de trabajo, y será más fina o más gruesa si el volumen de trabajo es menor o mayor. Podemos decir que es el tamaño del conjunto de operaciones que pueden realizarse en paralelo.

Paralelismo de tareas vs paralelismo de datos No es lo mismo trabajar en paralelo únicamente con datos, únicamente con tareas, o mezclando ambos.

El paralelismo de tareas se encuentra extrayendo la estructura lógica de funciones. Es por esto que está relacionado con el paralelismo *a nivel de función*. El paralelismo de datos se encuentra implícito en las operaciones con estructuras de datos, y relacionado con el paralelismo a nivel del *bucle*.

1.1.2. Condiciones de paralelismo

Para poder realizar ejecuciones en paralelo entre dos bloques de código, deben presentar una independencia de datos.

Dependencia de datos Decimos que un bloque de código B_2 presenta dependencia de datos con respecto a otro bloque B_1 si:

- Hacen referencia a una misma posición de memoria `M` (variable).
- B_1 aparece en la secuencia de código antes que B_2.

Podemos clasificar la dependencia de datos en tres tipos, según la secuencia de operaciones de lectura/escritura que ocurran en los dos bloques con respecto a `M`.

RAW (Read After Write): dependencia verdadera. No se puede evitar.

WAW (Write After Write): dependencia de salida. Podría evitarse guardando la variable inicial en una variable temporal.

WAR (Write After Read): antidependencia. Podría evitarse realizando la lectura en una variable temporal que contuviese la variable inicial.

El caso *RAR (Read After Read)* no presenta dependencia, ya que simplemente se consulta el dato, sin modificarlo en ningún caso.

1.1.3. Paralelismo explícito

Un computador puede aprovechar el paralelismo entre diferentes entidades de forma explícita, a saber:

- Instrucciones.
- Hebras (*threads*).
- Procesos.

Hebras vs procesos Un **proceso** comprende el código del programa, y todo lo necesario para su ejecución (datos en pila, segmentos, registros, tabla de páginas, ...). Para comunicar procesos hay que usar llamadas al SO.

Un proceso puede constar de múltibles **hebras** que controlan el flujo de control. Cada hebra tiene su propia pila, y también el contenido de los registros. Para comunicarse entre ellos, utilizan la memoria que comparten.

En general, las operaciones de creación, destrucción, conmutación y comunicación en las hebras consumen menos tiempo.

1.2. Lección 2. Clasificación de estructuras paralelas

1.2.1. Computación paralela y computación distribuida

La **computación paralela** estudia el desarollo y ejecución de aplicaciones en un sistema compuesto por múltiples cores/procesadores que es visto externamente como una unidad autónoma.

Mientras que la **computación distribuida** estudia los aspectos del desarrollo y ejecución de aplicaciones en un sistema distribuido, es decir, en una colección de recursos autónomos situados en distintas localizaciones físicas.

Esta última puede ser a **baja escala**, si se encuentran situados en diversas localizaciones unidas por una misma infraestructura de red local. O bien **computación grid**, un conjunto de dominios administrativos distribuidos en distintas localidades geográficas y que están unidas con una infraestructura de telecomunicaciones.

Ejemplos de esto son la *computación cloud*, que se desarrollan en un *sistema cloud*, un conjunto de recursos virtuales (pay-per-use y con interfaz de autoservicio) que abstraen los recursos físicos utilizados, con recursos que son cap-

tados/liberados de manera inmediata, con acceso múltiple de clientes y con métodos de conexión estándar independientes de la plataforma de acceso.

1.2.2. Criterios de clasificación de computadores

Por lo general la clasificación de computadores se basa en el número de núcleos que posean y operaciones que sean capaces de hacer, lo cual influye directamente en el precio.

Clasificación de Flynn Se basa en el flujo de instrucciones, si son capaces de trabajar con una única instrucción o con varias y si son capaces de trabajar con un único flujo de datos o con varios.

SISD Single Instruction, Single Data. Existe un único flujo de datos y una única instrucción a ejecutar en cada instante. Corresponde a los computadores uni-procesador. **<!- (algo de único flujo de datos y una única instrucción, completar) –>**

SIMD Single Instruction, Multiple Data. La instrucción que se codifica va a cada uno de los procesadores (es la misma para todos), donde se le indica de dónde tiene que captar los datos, hace tantas operaciones como unidades de procesamiento se tengan. Por tanto, aprovecha el paralelismo de datos.

MISD Multiple Instruction, Single Data. No es un sistema que se implemente en la realidad, cada unidad de control estaría conectada con una única unidad de procesamiento con cada flujo de datos.

MIMD Multiple Instruction, Multiple Data. Múltiples flujos de datos que permiten realizar múltiples instrucciones. Aprovecha, además del paralelismo de datos, el paralelismo funcional. **<!- (algo de múltiples flujos de datos y múltiples instrucciones, completar) –>**

Sistema de memoria Existen dos tipos de máquinas según esta clasificación: multiprocesadores y multicomputadores.

En los **multiprocesadores** todos los procesadores comparten el mismo espacio de direcciones, el programador no necesita conocer dónde están almacenados los datos. Tienen mayor latencia y son poco escalables. Tienen comunicación implícita mediante variables compartidas y los datos no duplicados están en memoria principal. Es necesario implementar primitivas de sincronización, no es necesaria la distribución de código y datos entre procesadores y su programación es mas sencilla.

En los **multicomputadores** cada procesador tiene su propio espacio de direcciones. Están compuestos por computadores completos conectados entre sí

por una interfaz de red. El programador necesita conocer donde están almacenados los datos. Tienen menor latencia y son escalables. Tienen comunicación explícita mediante software para paso de mensajes y los datos duplicados están en memoria principal. Su sincronización es mediante un software específico de comunicación, es necesaria la distribución de código y datos entre procesadores y su programación es más difícil.

Comunicación uno-a-uno en un multiprocesador El nodo fuente manda a memoria el dato, se procesa y se genera una respuesta que es devuelta al nodo fuente. Después, el nodo destino realiza la petición del dato a memoria, se procesa y la memoria devuelve el dato al flujo destino. Se debe garantizar que el flujo de control *consumidor* del dato lea la variable compartida cuando el *productor* haya escrito en la variable el dato.

Comunicación uno-a-uno en un multicomputador El nodo fuente copia los datos que desea enviar a un buffer y mediante la red de interconexión los hace llegar al nodo destino que está en ejecución a la espera de que le lleguen los datos. Cuando llegan, los datos se cargan en memoria de usuario y continua con la ejecución.

Incremento de escalabilidad en multiprocesadores Se debe aumentar la caché del procesador, usar redes de menor latencia y mayor ancho de banda que un bus y distribuir físicamente los módulos de memoria entre los procesadores compartiendo el espacio de direcciones.

Clasificación completa de computadores según el sistema de memoria

Arquitecturas con DLP, ILP y TLP *(Thread=Flujo de control)*

1.3. Lección 3. Evaluación de prestaciones de una arquitectura

1.3.1. Tiempos de CPU

Tiempo de CPU Tcpu = Ciclos*Tciclo = Ciclos/Frecuencia de reloj

Ciclos por Instrucción CPI = Ciclos/NI

Donde NI es el nž de instrucciones y donde los ciclos son la suma de los CPI de cada instruccion. Así vemos que hay varias formas de calcular el tiempo de CPU:

Tcpu = NI * (CPE / IPE) * T1ciclo

Tcpu = NI * CPI * Tžciclo

Tcpu = (Nžoperaciones/OPI) * CPI * Tžciclo

OPI: Número de operaciones que puede codificar una instruccion

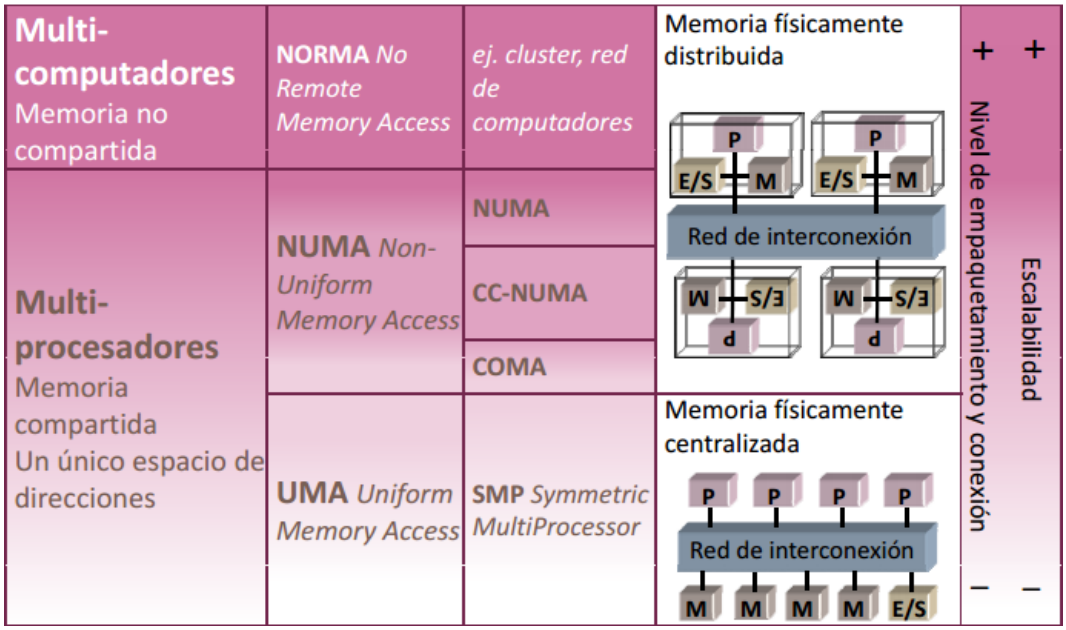


Figura 1: Clasificación

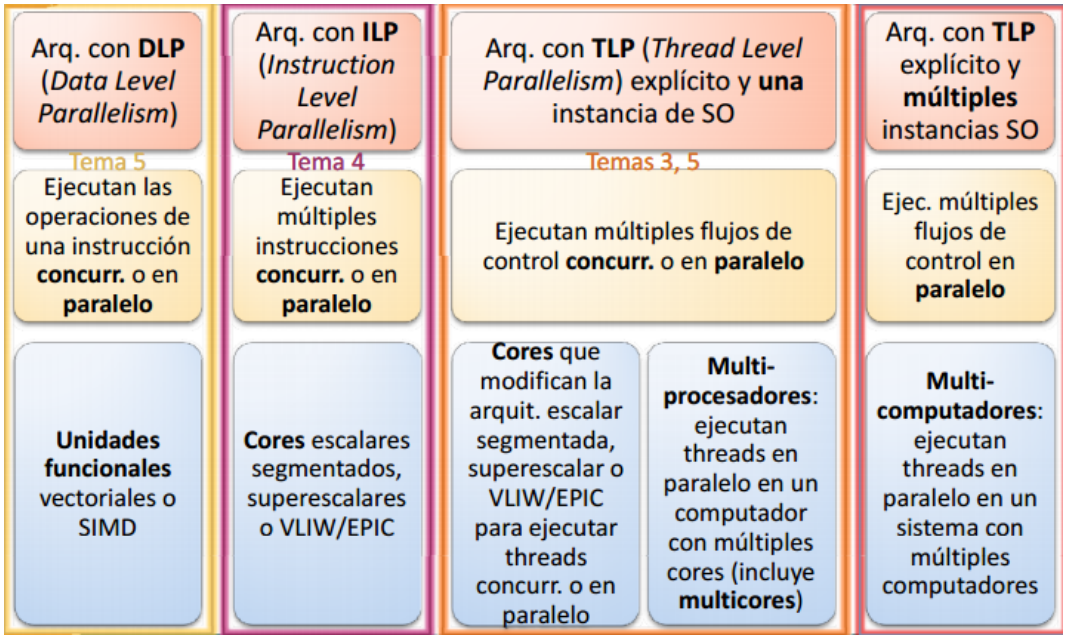


Figura 2: Arquitecturas

CPE: Número mínimo de ciclos transcurridos entre los instantes instantes en que el procesador procesador puede emitir instrucciones instrucciones

IPE: Instrucciones que pueden emitirse (para empezar su ejecución) cada vez que se produce dicha emisión

MIPS MIPS: Millones de intrucciones por segundo

MIPS=NI/(Tcpu * 10^6)=Frecuencia/(CPI * 10^6)

Depende del repertorio de instrucciones. Puede variar con el programa. Puede variar inversamente con las prestaciones (mayor valor de MIPS corresponde a peores prestaciones)

MFLOPS MFLOPS: Millones de operaciones en coma flotante por segundo

MFLOPS= Operaciones en coma flotante/(Tcpu * 10^6)

No es una medida adecuada para todos los programas. El conjunto de operaciones en coma flotante no es constante en máquinas diferentes y la potencia de las operaciones en coma flotante no es igual para todas las operaciones

1.3.2. Conjunto de programas de prueba (Benchmark)

Tipos

- De bajo nivel o microbenchmark
- Kernels
- Sintéticos
- Programas reales
- Aplicaciones diseñadas

1.3.3. Ganancia en prestaciones

Speed-Up El incremento de velocidad que se consigue en la nueva situación con respecto a la previa (máquina base) se expresa mediante la ganancia de velocidad o speed-up:

Sp = Vp / V1 = T1 / Tp

V1 Velocidad de la máquina base Vp Velocidad de la máquina mejorada (un factor p en uno de sus componentes) T1 Tiempo de ejecución en la máquina base Tp Tiempo de ejecución en la máquina mejorada

Ley de Amdahl La mejora de velocidad, S, que se puede obtener cuando se mejora un recurso de una máquina en un factor p está limitada por:

$S \leq p/(1 + f(p-1))$

donde f es la fracción del tiempo de ejecución en la máquina sin la mejora durante el que no se puede aplicar esa mejora.

Lo vemos con un ejemplo:

Si un programa programa pasa un 25% de su tiempo de ejecución en una máquina realizando instrucciones de coma flotante, y se mejora la máquina haciendo que estas instrucciones se ejecuten en la mitad de tiempo, entonces p=2, f=0.75 y $S \leq 2/(1+075) = 1.14$

qHay que mejorar el caso más frecuente (lo que más se usa)!

Ley enunciada por Amdahl en relación con la eficacia de los computadores paralelos: dado que en un programa hay código secuencial que no puede paralelizarse, los procesadores no se podrían utilizar eficazmente

2. Tema 3

2.1. Arquitecturas TLP

Las arquitecturas con TLP(Thread Level Parallelism) son:

2.1.1. Multiprocesador:

Varios threads en paralelo en un **computador** con varios procesadores, cada thread en un core/procesador distinto.

Hay dos tipos:

1. Memoria centralizada (*UMA*: Uniform Memory Access), en el que todos los procesadores tienen acceso a toda la memoria mediante una red de interconexión. Tienen una mayor latencia y son poco escalables.
2. Memoria distribuida (*NUMA*: Non Uniform Memory Access), en el que cada procesador tiene su zona de memoria y se conectan mediante una red de interconexión. Tienen menor latencia y son escalables, pero requieren para ello distribución de datos y código.

2.1.2. Multiprocesador en un chip o multicore:

Ejecutan varios threads en paralelo en un **chip de procesamiento** multicore (cada thread en un core distinto). Existen distintas posibilidades en cuanto a la distribución de las cachés: + Los cores tengan cachés independientes y /todos/ tengan alguna caché común. + Agrupados para que todos los cores del mismo grupo tengan (a parte de sus cachés independientes) alguna caché común. + No tengan ninguna caché común y se conectan directamente al conmutador

2.1.3. Core multithread:

Modifican su estructura ILP(Instruction level parallelism) para ejecutar threads concurrentemente o en paralelo.

1. Los procesadores segmentados ejecutan instrucciones concurrentemente segmentando el uso de sus componentes.
2. Los procesadores VLIW(very large Instruction Word) y superescalares ejecutan instrucciones concurrentemente (segmentación) y en paralelo(emitiendo múltiples instrucciones a sus múltiples unidades funcionales).

- VLIW:
  - Las instrucciones que se ejecutan en paralelo se captan juntas en memoria.
  - Este conjunto de instrucciones conforman la palabra de instrucción muy larga a la que hace referencia la denominación VLIW.
  - El hardware presupone que las instrucciones de una palabra son independientes: no tiene que encontrar instrucciones que pueden emitirse y ejecutarse en paralelo.
- Superescalares:
  - Tienen que encontrar instrucciones que puedan emitirse y ejecutarse en paralelo (disponen de hardware para extraer paralelismo a nivel de instrucción).

2.1.4. Clasificación de cores multithread:

a. Temporal multithreading(TMT): Un core ejecuta varios threads, y la comunicación entre éstas la controla el hardware, emitiendo instrucciones de un único thread por ciclo. A su vez, estos pueden ser:
   1. Fine-grain multithreading, la conmutación entre threads la hace el hardware sin coste, se hacen por turno rotatorio y por eventos de cierta latencia combinado con alguna técnica de planificación.
   2. Coarse-grain multithreading, la conmutación la decide el hardware con algún coste (varía entre 0 y varios ciclos), tras intervalos de tiempo prefijados o por eventos de cierta latencia. Pueden ser :
      - Estáticos: tienen instrucciones explícitas para conmutar e implícitas (instrucciones de carga, almacenamiento, salto) y cambio de coste bajo como ventaja, pero produce cambios de contexto innecesarios.
      - Dinámicos: conmutan cuando hay fallos de caché o interrupciones. Tiene como ventaja que reduce los cambios de contexto innecesarios pero aumenta la sobrecarga al cambiar de contexto

b. Multihilo simultáneo(SMT): se ejecutan en un core superescalar varios threads en paralelo, emitiendo instrucciones de varios threads en un ciclo.

2.2. Coherencia del sistema de memoria

2.2.1. Sistema de memoria en multiprocesadores

El sistema de memoria incluye cachés, memoria principal, controladores, buffers y comunicaciones entre estas componentes. Este sistema se encarga de comunicar los datos entre los procesadores.

2.2.2. Coherencia en el S.M.

Los datos modificables pueden causar problemas con la E/S, los datos modificables compartidos pueden dar fallos de caché o lecturas de caché no actualizadas, y los privados si emigra el proceso dan también fallos de caché.

La memoria se puede actualizar de forma *inmediata*(write-through), cada vez que un procesador escribe en su caché lo hace también en memoria principal, lo cual no es muy rentable; o mediante *posescritura*(write-back), si se escribe todo el bloque cuando se desaloja de la caché.

Para intentar hacer la caché más coherente, se puede intentar hacer una *escritura con actualización*(write-update), en la que si un procesador escribe en una dirección de su caché, se escribe en las copias de esa dirección en las otras cachés, o una *escritura con invalidación*, en la que antes de modificar una dirección en su caché se invalidan las copias del bloque de la dirección en otras cachés.

Requisitos del sistema para evitar problemas por incoherencia

1. Propagar las escrituras en una dirección, haciendo esta escritura visible a otros procesadores teniendo los componentes conectados por un bus. Ahora, si la red no fuera un bus, las actualizaciones se envían a todas las cachés y para mejorar la escalabilidad se envían las actualizaciones sólo a las cachés que tengan copia de ese bloque.
2. Serializar las escrituras en una dirección, viéndose en el mismo orden por todos los procesadores con los componentes conectados por un bus. Si la red no es un bus, el orden en el que las peticiones de escritura llegan al nodo que tiene en MP la dirección sirve para serializar en sistemas de comunicación que garantizan el orden de las transferencias.

El *directorio de memoria principal* está dentro de un nodo de cómputo y tiene un vector de bits con información sobre cachés con copia y un bit para el estado del bloque de memoria. Este se puede implementar de forma:

- Centralizada: Compartido por los nodos, contiene información de los bloques de todos los módulos de memoria.
- Distribuida: Las filas se distribuyen entre los nodos, y el directorio de cada nodo tiene información de los bloques de /sus/ módulos de memoria.

2.2.3. Protocolos de mantenimiento de coherencia

Los protocolos utilizados para mantener la coherencia de memoria son de tres tipos: - Protocolos de espionaje (snoopy). Para sistemas con una difusión eficiente (buses, número pequeño de nodos o red con difusión). - Protocolos basados en directorios. Para redes sin difusión o escalables. - Esquemas jerárquicos. Para redes jerárquicas: jerarquía de buses, jerarquía de redes escalables, redes escalables-buses.

Existen diferentes facetas a tener en cuenta a la hora de diseñar un protocolo para mantener la coherencia de memoria. Estas son:

- Política de actualización de memoria principal: escritura inmediata, posescritura, mixta.
- Política de coherencire caches: escritura con invalidación, escritura con actualización, mixta.
- Describir compotamiento: Definir posibles estados de los bloques en cache y memoria, definir transferencias a genenrar entre eventos y definir transiciones de estados para un bloque en cache y en memoria.

Protocolo de espionaje de tres estados MSI

- Estados de un bloque en cache:
  - Modificado (M): es la única copia del bloque válida en el sistema.
  - Compartido (C, S): está válido, también válido en memoria y puede que haya copia válida en otras caches.
  - Inválido (I): se ha invalidado o no está físicamente.
- Estado de un bloque en memoria:
  - Válido: puede haber copia válida en una o varias caches.
  - Inválido. habrá copia válida en una caché.
- Transferencias generadas por un nodo de cache:
  - Peticion de lectura de un bloque (PtLec): por lectura con fallo de cache del procesador del nodo (PrLec).
  - Petición de acceso exclusivo (PtLecEx): por escitura del procesador (PrEsc) en bloque compartido o inválido.
  - Petición de posescritura (PtPEsc): por el reemplazo del bloque modificado.
  - Respuesta con bloque (RpBloque): al tener en estado modificado el bloque solicita una PtLec o PtLecEx recibida.

Protocolo de espionaje de cuatro estados MESI

- Estado de un bloque en cache:

  - Modificado (M): es la única copia del bloque válida en todo el sistema.
  - Exclusivo (E): es la única copia válida del bloque en caches, la memoria también está actualizada.
  - Compartido (C, S): es válido, también es válido en memoria y al menos otra cache.
  - Inválido(I): se ha invalidado o no está físicamente.

- Estado de un bloque en memoria:

  - Válido: puedes haber una copia válida en una o varias caches.
  - Inválido: habrá una copia válida en una cache.

Protocolo MSI con directorios sin difusión

- Estado de un bloque de cache:

  - Modificado (M)
  - Compartido (C)
  - Inválido (I)

- Estados de un bloque en MP:

  - Válido
  - Inválido

- Transferencias: Tipos de nodos: solicitante (S), origen(O), modificado (M), propietario (P) y compartidor (C.

  - Petición de nodo S a O: lecura de un bloque (PtLec), lectura con acceso exclusivo (PtLecEx), petición de acceso exclusivo sin lectura (PtEx), posescritura (PtPEsc).
  - Reenvío de petición de nodo O a nodo copia (P, M, C): invalidación (RvInv), lectura (RvLec, RvLecEx).
  - Respuesta de:

    Nodo P a O: respuesta con bloque (RpBloque), respuesta con o os in bloque confirmando fin de invalidez (RpInv, RpBloqueInv).
    Nodo O a S: respuesta con bloque (PrBloque), respuesta con o sin bloque confirmando fin de invalidez (RpInv, RpBloqueInv).

2.2.4. Consistencia de memoria

Especifica las restricciones en el orden en el cual las operaciones de memoria (lectura y escritura) deben parecer haberse realizado. La coherencia sólo abarca operaciones realizadas por múltiples componentes en una misma dirección. En un procesador (o sistema uniprocesador), el orden en el que deben parecer haberse ejecutado los accesos a memoria es el orden secuencial especificado por el programador, denominado **orden del programa**.

Consistencia secuencial Todas las operaciones de un único procesador parecen ejecutarse en el orden descrito por el programa de entrada al procesador. Todas las operaciones de memoria parecen ser ejecutadas una a la vez. Presenta el sistema de memoria a los programadores como una memoria global conectada a todos los procesadores a través de un conmutador central. Informalmente, diríamos que el modelo de consistencia secuencial require que todas las operaciones de memoria parezcan ser ejecutadas *una vez* (ejecución *atómica*) y que las instrucciones de un único porcesador (proceso) parezcan ejecutarse en el *orden descrito por el programa* de entrada al procesador.

Modelos de consistencia relajados Relajan requisitos (orden, atomicidad) de consistencia de memoria para aumentar prestaciones. + **Orden del programa**. Los modelos pueden permitir que en el código ejecutado en un procesador se relaje en el orden entre dos accesos a distintas direcciones. Difieren en cuanto a los órdenes que permiten relajar; pueden permitir alterar el orden entre escrituras (W->W), entre lecturas (R->R), entre una lectura y una escritura posterior (W->R) o entre una escritura y una lectura posterior (W->R). + **Atomicidad**. Hay modelos que permiten que un procesaodr pueda leer el valor escrito por otro procesador antes de que esta escritura se haga visible para el resto de procesadores.

Modelo que relaja W->R Permiten que una lectura pueda adelantar a una escritura precia en el orden del programa, pero evita dependencia RAW. Lo implementan los sistemas con buffer de escritura para los procesadores. Generalmente permiten que el procesador pueda leer una dirección directamente del buffer. Hay sistemas en los que se permite que un procesador pueda leer la escritura de otro antes que el resto.

Modelo que relaja W->R y W->W Tiene buffer de escritura que permite que lecturas adelanten a escrituras en buffer. Permiten que el hardware solape escrituras a memoria en distintas direcciones, de forma que pueden llegar a la memoria principal o a caches de todos los procesadores fuera del orden del programa.

Modelo de ordenación débil Relaja W->R, W->W y R->W.

2.2.5. Sincronización

Cerrojos Permiten sincronizar mediante dos operaciones: - Cierre del cerrojo (lock(k)): intenta adquirir el derecho a acceder a una sección crítica. Si varios procesos intentan adquisición a la vez, solo uno de ellos lo debe conseguir, el resto debe pasar a una etapa de espera. Todos los procesos que ejecuten lock() con el cerrojo cerrado deben quedar en espera. - Apertura del cerrojo (unlock(k)): libera a uno de los threads que esperan el acceso a una sección crítica. SI no hay

threads en espera, permitirá que el siguiente thread que ejecute lock() adquiera el cerrojo k sin espera.

Componentes en un código para sincronización

- Método de adquisición. Método por el que un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas.
- Método de espera. Método por el que un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas.
- Método de liberación. Método utilizado por un thread para liberar uno o varios threads en espera.

3. Tema 4

3.1. Lección 11. Microarquitecturas ILP. Cauces superescalares

Como ya sabemos, los procesadores superescalares y VLIW nos permiten obtener una mayor eficiencia a la hora de procesar cualquier tarea. Ambos tienen varias unidades de ejecución, pueden ejecutar varias instrucciones simultáneamente en dichas unidades y además pueden emitir múltiples instrucciones en paralelo a estas.

Su principal diferencia radica en que en los procesadores superescalares es el hardware el que debe descubrir el paralelismo en tanto a las instrucciones que se van captando mientras que los VLIW requieren un paralelismo explícito (se captan las instrucciones que se van a emitir juntas a unidades de ejecución).

La micro-arquitectura de los procesadores VLIW es más sencilla, pues es el compilador el que debe detectar el paralelismo al seleccionar las instrucciones que se captarán juntas en la misma palabra de instrucción.

3.1.1. Paralelismo entre instrucciones. Orden en Emisión y Finalización

ILP(Instruction Level Paralelism): Depende de la frecuencia de las dependencias de datos y control, y del retardo de la operación (tiempo hasta que el resultado de una operación esté disponible para otra operación que espere dicho resultado).

Paralelismo de máquina: Viene determinado por el número de instrucciones que pueden captarse y ejecutarse al mismo tiempo y por la velocidad y mecanismos que usa el procesador para encontrar las dependencias entre instrucciones.

Existen tres tipos de ordenaciones en una secuencia de instrucciones:
-Orden en el que se captan las instrucciones
-Orden en el que se ejecutan

-Orden en el que se cambian los registros y la memoria

El procesador superescalar es el responsable de gestionar el paralelismo entre instrucciones, además de organizar todas estas ordenaciones con el fin de mejorar la eficiencia de la máquina. Su única restricción es que el resultado del programa sea correcto.

3.1.2. Cauces Superescalares

De manera análoga a las etapas en el cauce de los procesadores normales, los superescalares tienen que dar cabida a la finalización de más de una instrucción por ciclo, esto implica que implementan otras etapas que se lo permiten.

Decodificación, emisión y ejecución paralela, finalización del procesamiento, detección y predicción de saltos y mantenimiento de la consistencia secuencial son los principales aspectos de estos cauces.

Predecodificación: Es la decodificación paralela que acabamos de mencionar. Añade bits para catalogar las instrucciones en función de si son de salto, si referencia a memoria o según el tipo de unidad funcional; con el fin de organizar para su posterior tratamiento específico.

Emisión paralela: A continuación las instrucciones deben ser enviadas a procesar. La **ventana de instrucciones** nos permite almacenar las instrucciones pendientes de tratar (puede ser centralizada o distribuida según si mezcla diversos tipos de instrucciones o no), se almacenan en ella una vez decodificadas y almacena en bits la disponibilidad de sus operandos y la unidad funcional donde se procesará para ser emitida cuando todo esté listo.

Podemos hacer una distinción ademas entre **emisión ordenada** si las instrucciones se empiezan a ejecutar en el mismo orden que en el programa, y **emisión desordenada** cuando pueden no respetar este orden.

En los procesadores superescalares se tiende a usar **estaciones de reserva** para distribuir la ventana de instrucciones. Estas son menos complejas a nivel de hardware, con un acceso más rápido ocupan menos líneas de memoria. En ellas las instrucciones esperan a que se resuelvan las dependencias.

Renombramiento de Registros: Para evitar el efecto de las dependencias WAR y WAW se pretende asignar cada escritura a un registro físico distinto. Esto puede ser realizado durante el proceso de compilación, pero en los procesadores superescalares se suele implementar en el hardware, mediante el uso de **buffers de renombramiento**, que son consultados al captar los operandos para conocer si se han escrito en otros registros y que no ha habido renombramiento.

3.2. Lección 12. Consistencia del procesador y Procesamiento de saltos

3.2.1. Consistencia. Reordenamiento

En el procesamiento de una instrucción podemos distinguir entre el *fin de la ejecución de la operación* y el *final del procesamiento de la instrucción*. La diferencia entre ambas es que la segunda no incluye la modificación de los registros, sólo la generación de resultados por las unidades funcionales.

La **consistencia** se define como el orden en el que las instrucciones se completan y el orden en el que se accede a memoria para leer o escribir. Así pues, para que un programa se ejecute correctamente debe haber consistencia entre el código del programa y el orden en el que se complletan las instrucciones.

La consistencia del *procesador* puede ser fuerte o débil. Débil permite la alteración del orden de las instrucciones siempre y cuando las dependencias no se vean afectadas. Fuerte por su parte exige un cumplimiento estricto del orden de ejecución con respecto al código del programa.

La consistencia de *memoria* también realiza esta distinción, permitiendo desordenar cuando se respeten las dependencias o exigiendo el acceso en un orden estricto.

Esta consistencia se puede implementar con Buffers de Reordenación (ROB), reordenando las instrucciones de lectura y escritura.

3.2.2. Procesamiento especulativo de saltos

Las instrucciones de salto condicional tienen unos efectos muy perjudiciales para los segmentados procesadores superescalares. Esto implica que se pueden producir ejecuciones innecesarias y que eso lleve a otras instrucciones que arreglen los efectos indeseados.

Las **instrucciones de salto incondicional** suponen un menor problema para el procesador, pues en la predecodificación se podrían marcar esas instrucciones para calcular la dirección del salto con antelación. Pero en el caso de las **instrucciones de salto condicional** no se puede calcular la dirección de salto hasta que no se evalue el objeto de la condición. Los procesadores intentan realizar una **predicción del salto** en el caso de que se cumpla alguna condición, generalmente la más probable, luego se comprueba si la predicción fue correcta, y con esto se pretende minimizar el coste de la evaluación o de la subsanación del error.

Existen diversos tipos de predicción. Estática (basada en código de operación, desplazamiento del salto o dirigida por el compilador), o dinámica (explícita, con usos de bits de historia, o implícita, sin estos, sólo almacena la dirección de la instrucción que se ejecutó después de la dirección de salto).

3.3. Lección 13. Procesamiento VLIW

Parte I.
Seminarios

4. Seminario 0. Atcgrid y TORQUE

Para estas prácticas hacemos uso de un servidor *front-end*:**Atcgrid**. Este tiene un conjunto de nodos de cómputo, a los cuales les envía las tareas de cálculo para obtener respuesta a las peticiones de los usuarios.

Esto es posible mediante **TORQUE**, un gestor de colas y recursos.

4.1. Comandos para TORQUE

Usamos comandos en atc mediante **ssh** para poder realizar los trabajos que queramos en él. Además, para subir nuestros archivos al front-end necesitamos comunicarnos con el servidor mediante **sftp**. Siempre debemos de estar conectados a la VPN de la ugr (usar eduroam o bien conectarse a la VPN con openconnect).

- psbnodes: Información sobre los nodos.
- qsub: Enviar un trabajo a ejecutar, devuelve la salida en un fichero output(*.o) y otro error (*.e).

Ejemplos de uso:

- “echo ‘hello’ | qsub -q ac” → Envía el trabajo(programa) hello por la cola ac
- “qstat” → Muestra los trabajos ejecutándose y los que están en las colas
- “echo ‘hello/trabajoMiTrabajo’ | qsub -q ac -N “NombreMiNombre” "

5. Seminario 1. Directivas OPENMP

Openmp es una API, capa de abstracción que permite al programador trabajar a través de una interfaz para aplicar el paralelismo en sus programas. Comprende un conjunto de directivas del compilador, funciones de biblioteca y variables de entorno.

Una **directiva** es "una marca" en nuestro archivo fuente que es sustituida por el preprocesador del compilador por otro código que permite realizar una tarea determinada sin tener que definirla nosotros explícitamente.

5.1. Sintaxsis de las directivas C/C++

```
1 #pragma omp <nombre_directiva> [<cláusula(s)>] <\n(newline)>
```

Donde:

- #pragma omp es necesario para indicar que estamos usando una directiva de openmp.
- es el nombre de la acción que realiza la directiva.
- es opcional, modifica o aporta información para la ejecución de la directiva, pueden combinarse.
- <\n(newline)> es el salto de línea necesario.

Compilación: Compilamos usando gcc -fopenmp para poder usar estas directivas.

5.2. Directivas

5.2.1. Parallel

Especifica qué cálculos se realizarán en paralelo. Un thread master crea un conjunto de threads cuando llega a esta directiva. El código contenido en esta región es ejecutado por cada thread. No reparte las tareas entre threads, tiene una barrera implícita y se pueden usar de forma anidada.

```
1 #pragma omp parallel
2 {
3     //Code
4 }
```

5.2.2. Trabajo compartido. Worksharing

Paralelismo de datos(for): Para distribuir las iteraciones de un bucle entre los diversos hilos usamos:

```
#pragma omp for
{
    //Code
}
```

Paralelismo de tareas(sections): Para distribuir trozos de código que son independientes entre sí.

```
#pragma omp sections
{
    #pragma omp section
    {
        //Codeblock 1
    }

    #pragma omp section
    {
        //Codeblock 2
    }
}
```

Ejecución única(single): Para que sólo un hilo ejecute un trozo de código (lo cual interesa por ejemplo para pedir/mostrar datos al usuario del programa una única vez en una situación de paralelismo).

```
#pragma omp single
{
    //Code
}
```

5.2.3. Combinando parallel con worksharing

Es posible, además del uso de las directivas de worksharing tras el uso de la directiva parallel, realizar una versión combinada única. Esto difiere del original tanto en legibilidad como en prestaciones.

```
#pragma omp parallel for
{
    //For loop
}
```

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        //Codeblock 1
    }

    #pragma omp section
    {
        //Codeblock 2
    }
}
```

5.2.4. Directivas básicas de comunicación y sincronización

En diversas ocasiones nos interesa que la *lectura/escritura* de una variable se hiciese en exclusión mutua (secuencial) para evitar que se modifique o use un valor de manera incorrecta. Para esto veremos las directivas **critical y atomic** y la directiva de "control" **barrier**.

Barrier: Es una barrera que se situa en el punto en el que esperamos que todos los threads lleguen ahí. Lo usamos cuando necesitamos todos los cálculos de los threads hasta ese punto para que no se produzcan errores.

Critical: Evita que varios threads accedan a variables compartidas a la vez (situaciones de carrera). Un thread protege una variable frente a los accesos de otros threads a la misma variable.

Atomic: Da una respuesta más eficiente que "*Critical*".

5.2.5. Directiva master

Es similar a la directiva single, pero en este caso la hebra que ejecutará el bloque de código será la *thread master* o "*hebra 0*".

6. Seminario 2. Cláusulas OpenMP

Las **cláusulas** son las encargadas de ajustar el comportamiento de las directivas. No pueden ser usadas en directivas tales como: master, critical, barrier, atomic, flush, ordered o threadprivate.

6.1. Ámbito de los datos por defecto. Compartición de datos.

Es conveniente saber qué valores tomarán las variables dentro de una zona donde queramos hacer uso del paralelismo. Tenemos que tener claro si queremos que la memoria se comparta o que en cada thread se mantengan unos datos privados.

Por lo general las variables declaradas fuera de una región y las dinámicas son compartidas por los threads de la región, mientras que las variables declaradas dentro son privadas.

A excepción de esto nos encontramos los índices de los bucles *for* y las variables declaradas *static*, que son privados y "públicas" respectivamente.

6.1.1. Shared

```
#pragma omp parallel for shared(a,b,...,N)‘‘‘

Las variables indicadas por la lista son compartidas por los
threads. Hay que tener cuidado cuando un thread lea lo que
   otro
escribe en una variable de la lista.

En el siguiente ejemplo inicializamos un vector y realizamos
   una pequeña operación sobre sus componentes, independiente
   entre ellas (lo que hagamos con la componente *i* no afecta
    a las otras). La operación la paralelizamos, compartiendo
   el vector *a[]* entre ellas.
El ejemplo es meramente ilustrativo, la utilidad de usar
   shared en este caso es prácticamente nula.

~~~c
#include <stdio.h>
#ifdef _OPENMP
    #include <omp.h>
#endif
```

```
15
16
17 int main() {
18     int n = 7;
19     int a[n];
20
21     for(int i = 0; i < n; ++i)
22         a[i] = i+1;
23
24     //Paralelizamos las interaciones de los bucles entre las
          hebras, siendo a común a las hebras
25     #pragma omp parallel for shared(a)
26     for(int i = 0; i < n; ++i)
27         a[i] += i;
28
29     printf("Después del parallel for:\n");
30
31     for(int i = 0; i < n; ++i)
32         printf("a[%d] = %d\n",i,a[i]);
33 }
34 ~~~
35 La cláusula gana algo de sentido cuando la combinamos con la
      cláusula default, por lo que retomaremos este ejemplo
      cuando se explique dicha cláusula.
36
37 ### Private
38 ```c
39 #pragma omp parallel for private(a,b,...,N)
```

De modo análogo a shared, indica una lista de variables cuya memoria no es compartida. Es importante saber que el valor de entrada y de salida están indefinidos aunque la variable haya sido definida antes de la región. Es decir, que por defecto, al entrar en la sección paralelizada no se sabe lo que vale, ni tampoco se sabe lo que valdrá al cerrar dicha zona. Por tanto, tenemos que definirla dentro (y si queremos, operar con ella), sabiendo así los valores que toma. Si no lo hacemos toma un valor basura, lo que hubiera en la dirección de memoria utilizada.

Los índices de los bucles tienen un ámbito predeterminado privado si se usa la directiva for, por lo que no hace falta especificarlo en la lista.

En el siguiente ejemplo, queremos que cada hebra i imprima i. Como declaramos el número de hebra como privado (nthread) y el valor que debe imprimir como compartido, nos aseguramos que la hebra imprima correctamente nthread, pero valor es compartido, por lo que puede que una hebra cuando vaya a impri-

mir valor, justo antes, otra hebra modifique el valor, perdiendo el que le había asignado la hebra.

Realizamos 3 iteraciones en cada hebra para poder apreciar algún fallo (puede darse que el resultado sea correcto, pero eso depende de las casuísticas a la hora de la ejecución).

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3      #include <omp.h>
4  #endif
5
6
7  int main() {
8      int nthread = 0;
9      int valor = 0;
10
11     #pragma omp parallel sections shared(valor) private(
           nthread)
12     {
13         #pragma omp section
14         {
15             nthread = omp_get_thread_num();
16             valor = nthread;
17
18             for(int i = 0; i < 3; ++i)
19                 printf("Soy la hebra %d, debo imprimir %d\n",
                       nthread, valor);
20         }
21
22         #pragma omp section
23         {
24             nthread = omp_get_thread_num();
25             valor = nthread;
26             for(int i = 0; i < 3; ++i)
27                 printf("Soy la hebra %d, debo imprimir %d\n",
                       nthread, valor);
28         }
29
30         #pragma omp section
31         {
32             nthread = omp_get_thread_num();
33             valor = nthread;
34             for(int i = 0; i < 3; ++i)
```

```
35                  printf("Soy la hebra %d, debo imprimir %d\n",
                        nthread, valor);
36          }
37
38          #pragma omp section
39          {
40              nthread = omp_get_thread_num();
41              valor = nthread;
42              for(int i = 0; i < 3; ++i)
43                  printf("Soy la hebra %d, debo imprimir %d\n",
                        nthread, valor);
44          }
45      }
46
47  }
```

Si ejecutamos el código anterior, observamos que algunas hebras imprimen un número que no se corresponde con ellas. Esto se debe a que otra hebra ha escrito antes de que la hebra actual imprima, produciéndose condiciones de carrera.

```
1  victor@victor-GL552VW:~/Documentos/Universidad/2ž Cuatrimestre
       /AC/Practicas/codi
2  gos_apuntes
       exportOMP_SET_NUM_THREADS=4victor@victor-GL552VW: /Documentos/Universidad/2C
        ./private
3  Soy la hebra 0, debo imprimir 0
4  Soy la hebra 0, debo imprimir 2
5  Soy la hebra 0, debo imprimir 2
6  Soy la hebra 2, debo imprimir 2
7  Soy la hebra 2, debo imprimir 2
8  Soy la hebra 2, debo imprimir 2
9  Soy la hebra 1, debo imprimir 1
10 Soy la hebra 1, debo imprimir 2
11 Soy la hebra 1, debo imprimir 2
12 Soy la hebra 3, debo imprimir 3
13 Soy la hebra 3, debo imprimir 2
14 Soy la hebra 3, debo imprimir 2
```

Si en lugar de declarar valor como shared, lo hacemos como private, el resultado de la ejecución es correcto.

6.1.2. Lastprivate

```
1  #pragma omp parallel for lastprivate(a,b,...,N)
```

Cada hilo tiene una copia local del dato. La copia global será actualizada por el hilo que ejecuta la última iteración según el orden secuencial de programa. Después de explicar la siguiente directiva veremos un ejemplo combinado

6.1.3. Firstprivate

```
1 #pragma omp parallel for firstprivate(a,b,...,N)
```

Combina la protección que otorga private pero al entrar en la región, en lugar de tener valores indefinidos, asigna los valores que tenía antes de entrar a cada variable de la lista para cada thread.

Cada hilo tiene una copia local del dato. La copia local se inicializa con el valor de la copia global en el momento de encontrarse con la directiva a la que se aplica la cláusula.

Útil para no olvidar la inicialización dentro de la región paralela cuando usamos variables private.

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3  #include <omp.h>
4  #else
5  #define omp_get_thread_num() 0
6  #endif
7  int main() {
8      int i, n = 7;
9      int a[n], suma=0;
10     for (i=0; i<n; i++)
11         a[i] = i;
12     #pragma omp parallel for firstprivate(suma) lastprivate(
           suma)
13         for (i=0; i<n; i++)
14         {
15             suma = suma + a[i];
16             printf(" thread %d suma a[%d] suma=%d \n",
17             omp_get_thread_num(),i,suma);
18         }
19         printf("\nFuera de la construcción parallel suma=%d\n"
               ,suma);
20 }
```

En el seminario, se plantea si siempre se imprime el mismo valor fuera de la región parallel. Al declarar suma como firstprivate, cada hebra se inicializa a 0, ya que es el valor que tenía antes de entrar a la región paralelizada. Al declararla también como lastprivate, el valor que le asigne la última hebra a suma será el

que se imprima fuera. Una explicación más detallada:

Lo que se imprime fuera de la región parallel, va a ser el valor suma que tenga la última hebra que realiza ejecuciones en orden secuencial. Como tenemos 7 iteraciones y 4 hebras disponibles, las 3 primeras hacen 2 iteraciones cada una, y la última hebra realiza la última iteración, que es un 6, por lo que la suma fuera del parallel se imprime un 6. Si ahora limitamos la ejecución a 3 hebras con export OMP_NUM_THREADS=3, la primera hebra hace 3 iteraciones, la segunda dos y la tercera, que es la última, también hace 2 iteraciones, las dos últimas. Estas dos últimas suman 5 y 6, 11 en total, por lo que la suma de la última hebra es un 11 y pone la suma global, la que está fuera del parallel, a 11, imprimiendo así un 11.

6.1.4. Default

Con `default(<none/shared>)` podemos alterar el comportamiento por defecto de las variables (sólo se puede usar una única vez. En caso de `none` habrá que especififcar el alcance de todas las variables usadas en la construcción por parte del programador.

Podemos excluir del ámbito por defecto usando todas las cláusulas de compartición que hemos visto hasta ahora.

Si se utiliza shared como default, estamos indicando que todas las variables que se utilicen en la sección paralelizada son compartidas por defecto. Si queremos que una en concreto es privada, tendremos que especificarlo con private. Por otro lado, si utilizamos none, hay que especificar el ámbito de todas las variables. Si no se hace con todas, el compilador producirá un error.

El siguiente código es el mismo que hemos utilizado para explicar la cláusula shared, pero situando delante default(none). Si lo compilamos, se produce un error, ya que no se declara el ámbito de la variable n. Para el índice del bucle no hay problema ya que como habíamos comentado, por defecto los índices de los bucles son privados al paralelizar con for.

Se soluciona cambiando shared(a) por shared(a,n). ~~~c #include <stdio.h> #ifdef _OPENMP #include <omp.h> #endif

int main() { int n = 7; int a[n]; int i;

```
1  for(int i = 0; i < n; ++i)
2      a[i] = i+1;
3
4  //Paralelizamos las interaciones de los bucles entre las
       hebras, siendo a común a las hebras
5  #pragma omp parallel for default(none) shared(a)
6  for(i = 0; i < n; ++i)
7      a[i] += i;
8
9  printf("Después del parallel for:\n");
```

```
10
11 for(i = 0; i < n; ++i)
12     printf("a[%d] = %d\n",i,a[i]);
```

} ~~~

6.1.5. Reduction

```
1 #pragma omp parallel for reduction(operator:list)
```

Esta cláusula indica que las variables de la lista serán tratadas según el operador indicado. Así de este modo se sumarán, restarán, multiplicarán... todas las variables del mismo nombre en distintos threads al final de la región tomando unos valores iniciales por defecto (el neutro para el correspondiente operador).

Es decir, si declaramos `reduction(+:suma)`, cada región paralelizada inicializa a 0 la variable suma (ver tabla inicializaciones abajo) y tras ejecutarse todas en paralelo, se suman todas las "sumas" que ha realizado cada hebra. Es como si tuviéramos una suma local para cada hebra y una suma global, donde a la suma global se le suma cada una de las locales. Al entrar en la región paralelizada cambia el valor al que dice la tabla, independientemente de lo que tuviera antes.

Operadores reduction (v3.0)

C/C++

tipo	Valor inicial
+	0
-	0
*	1
&	~0
|	0
^	0
&&	1
||	0

6.1.6. Copyprivate

```
1 #pragma omp parallel
2 {
3 // init list vars
4
5     #pragma omp single copyprivate(list)
6     {
7         //Codeblock
```

```
8       }
9   }
```

Esta cláusula solo se puede usar con la directiva single, y dentro de una región paralela copia el valor de la variable en el thread que ejecuta el single a la misma variable privada en los otros threads. Esto es usado comúnmente en lecturas o peticiones al usuario únicas.

En el siguiente ejemplo, la hebra que realiza el single inicializa el valor de la variable a. El resto de hebras tomarán dicho valor para la variable, que la tienen como privada, lo que haga cada hebra no afecta a otras.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5      int n = 9, i, b[n];
6      for (i=0; i<n; i++)
7      b[i] = -1;
8      #pragma omp parallel
9      { int a;
10         #pragma omp single copyprivate(a)
11         {
12         printf("\nIntroduce valor de inicialización a: ");
13         scanf("%d", &a );
14         printf("\nSingle ejecutada por el thread %d\n",
15         omp_get_thread_num());
16         }
17         #pragma omp for
18         for (i=0; i<n; i++) b[i] = a;
19     }
20     printf("Depués de la región parallel:\n");
21     for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
22     printf("\n");
23 }
```

7. Seminario 3. Variables de OpenMP

##Variables de control

1. *dyn-var* controla el ajuste dinámico del nž de threads.
2. *nthreads-var* controla el nž de threads en la siguiente ejecución paralela.

3. *thred-limit-var* controla el máximo numero de threads.
4. *nest-var* controla el paralelismo anidado.
5. *run-sched-var* controla la planificacion de bucles para *runtime*

##Variables de entorno

Se evitan con #ifdef_OPENMP ... (funciones)... #endif para asegurarnos de que sólo se usarán cuando estemos usando -fopenmp

1. *dyn-var* OMP_DYNAMIC export OMP_DYNAMIC=FALSE/TRUE
2. *nthreads-var* OMP_NUM_THREADS export OMP_NUM_THREADS=8
3. *thread-limit-var* OMP_THREAD_LIMIT export OMP_THREAD_LIMIT=8
4. *nest-var* OMP_NESTED export OMP_NESTED=TRUE/FALSE
5. *run-sched-var* OMP_SCHEDULE export OMP_SCHEDULE="static,4"/"dynamic"

##Funciones del entorno de ejecución

1. *dyn-var* omp_get_dynamic() omp_set_dynamic()
2. *nthreads-var* omp_get_max_threads() omp_set_num_threads()
3. *thread-limit-var* omp_get_thread_limit()
4. *nest-var* omp_get_nested() omp_set_nested()
5. *run-sched-var* omp_get_schedule(&kind, &modifier) omp_set_schedule(kind, modifier)

- omp_get_thread_num() - Devuelve al thread su identificador dentro del grupo de thread.
- omp_get_num_threads() - Obtiene el nž de threads que se están usando en una región paralela.
- omp_get_num_procs() - Devuelve el nž de procesadores disponibles para el programa en el momento de la ejecución.
- omp_in_parallel() - Devuelve true si se llama a la rutina dentro de una región parallel activa.

##Clausulas para interactuar con el entorno

Prioridad:

1ž Cláusula id 2ž Cláusula num_threads 3ž Función omp_set_num_threads() 4ž OMP_NUM_THREADS 5ž Prefijado en la implementación

###Cláusula if

Sintaxsis: if(condición)

No hay ejecución paralela si no se cumple la condición

###Cláusula schedule

Sintaxsis: schedule (kind[,chunk])

kind: static dynamic guided auto runtime

chunk: granularidad de la distribución

Solo se usa en bucles. Define el modo en el que se granula el trabajo

8. Arquitecturas con paralelismo a nivel de thread (TLP)

Ejercicio 8.1. Suponemos que se va a ejecutar en paralelo el siguiente código (inicialmente x e y son 0):

```
P1          P2
x=1;        y=1;
x=2;        y=2;
print y;    print x;
```

Qué resulados se pueden imprimir si (considere que el compilador no altera el código):

1. Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.
2. Se ejecutan en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden $W \rightarrow R$. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que se ejecuta adelanten a las escrituras que tiene su buffer. Obsérvese que hay varios resultados posibles.

Solución. Sabemos que `x = y = 0`

```
            P1          P2
(1) (W)     x=1;        y=1; (w) (a)
(2) (W)     x=2;        y=2; (w) (b)
(3) (R)     print y;    print x; (R) (C)
```

```
y=0  x=2  y=2  x=0
y=1  x=2  y=2  x=1
y=2  x=2  y=2  x=2
```

Ejercicio 8.2.

Ejercicio 8.3. Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 8 GBytes de memoria y una línea de cache supone 64 Bytes. Considere que el directorio utiliza vector de bits completo.

1. Calcule el tamaño del directorio de uno nodo en bytes.
2. Indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (inicialmente D no está en ninguna caché):

- Lectura generada por el procesador del nodo 1
- Escritura generada por el procesador del nodo 1
- Lectura generada por el procesador del nodo 2
- Lectura generada por el procesador del nodo 3
- Escritura generada por el procesador del nodo 0

Solución. La memoria principal es de 32Gb (4 nodos de 8Gb). Cada línea de caché (el tamaño de cada bloque de caché) tiene 64 bytes.

1. Teniendo en cuenta que hay una entrada por cada marco de bloque o de línea, tenemos que calcular cuantas líneas hay en cada memoria principal de cada nodo. Multiplicando por el número de bits de cada nodo, tendremos el tamaño que buscamos.

$$
\begin{aligned}
MPN &= 8GBytes = 2^3 \cdot 2^{30} = 2^{33}\ Bytes \\
LC &= 64\ Bytes = 2^6\ Bytes \\
\text{Nž líneas } &= \frac{2^{33}\ Bytes}{2^6\ Bytes} = 2^{27} \text{ líneas} \\
\text{Tamaño directorio memoria } &= 2^{27} l \cdot (4+1)\ bits = 5 \cdot 2^{27} bits \cdot \frac{1\ Byte}{2^3\ bits} = 5 \cdot 2^{24}\ Bytes \\
&= 5 \cdot 2^4 \cdot 2^{20}\ Bytes = 80\ MBytes
\end{aligned}
$$

2.