Multiprocesadores 3º Informática

Apellidos y nombre: Sergio García Esteban

1ª convocatoria, 15 junio 2020

Problema 1

Rendimiento de un código vectorizable en ZV — **20 puntos**

Estudiad el rendimiento de este código en un procesador vectorial ZV con 16 bancos de memoria. Cada banco contiene palabras de 8 bytes. Restricción: las lecturas con *stride* que tardan más de un ciclo en suministrar cada elemento NO encadenan.

R_A = 0x1000, R_B = 0x8000, R_C = 0xF000, R_S = 64, F_α = 3.1416

```
1$:  LV     V1,RA
     LVWS   V2,RB,RS
     MULSV  V3,V2,Fα
     ADDV   V4,V3,V1
     SV     RC,V4
```

8 elems

sección residual al final

código de seccionado

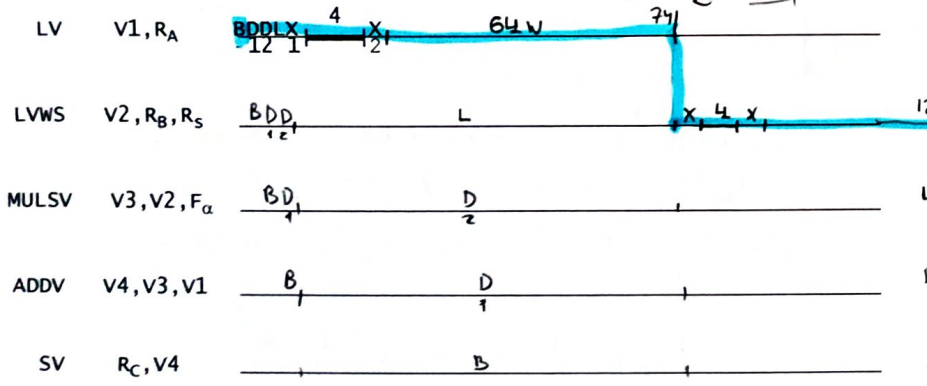
; 5 instrucciones escalares independientes
; salto no retardado, @dst se calcula en **D1**

a) Calcula los ciclos por elemento de las instrucciones LV y LVWS.

$P = \frac{M}{mcd(S,M)} \rightarrow P_{LV} = \frac{16}{1} = 16 \quad P_{LVWS} = \frac{16}{8} = 2$

$Ce_{LV} = \frac{L}{P} = \frac{4}{16} \simeq \boxed{1}$

$Ce_{LVWS} = \frac{L}{P} = \frac{4}{2} = \boxed{2}$



b) Diagrama de tiempos para la primera sección y media, señalando el principio de los encadenamientos y las detenciones por cualquier causa. Anota el ciclo de finalización de cada instrucción vectorial y de los puntos de interés.

c) Calcula Cn para n <= 64.

$Cn = 10 + n + 6 + 2n + 11 + n + 4 = \boxed{31 + 4n}$

d) Calcula Cn para n >> 64. Marca el camino crítico en el diagrama.

$Cn = \boxed{\lceil \frac{n}{64} \rceil \cdot 31 + 3n}$

e) Calcula R_∞

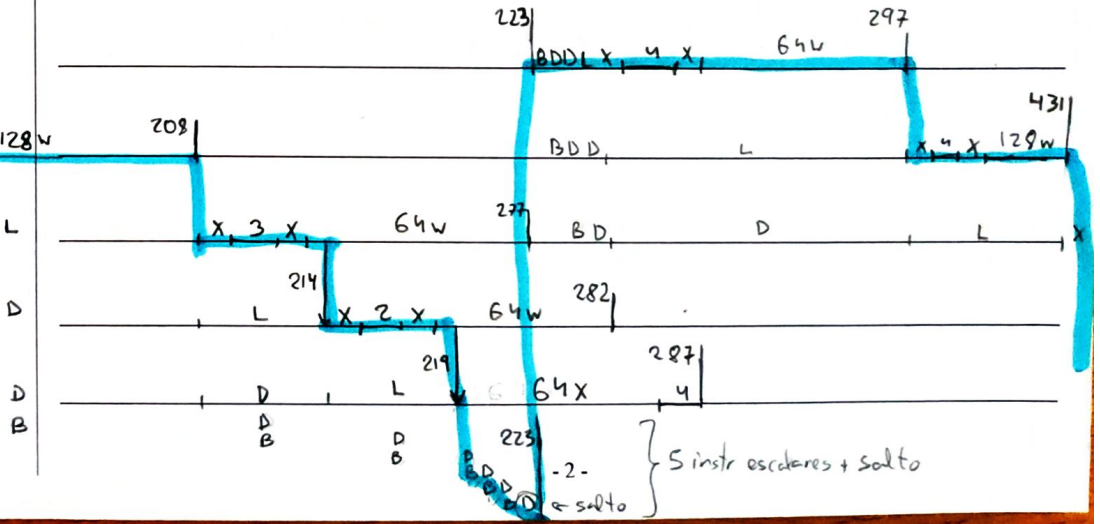
$R_\infty = \lim_{n\to\infty} \frac{2n}{\lceil \frac{n}{64} \rceil \cdot 31 + 3n} = \lim_{n\to\infty} \frac{128n}{223n} = \boxed{0{,}573 \text{ GFLOPS}}$

f) Calcula $N_{1/2}$

hipotesis: $N_{1/2} \leq 64$ → utilizamos $Cn_{(n\leq 64)}$

$\frac{2n}{31+4n} = \frac{0{,}573}{2} \rightarrow 4n = 17{,}79 + 2{,}29n$

$n = \frac{17{,}79}{4 - 2{,}29} = 10{,}43 \simeq \boxed{11 \text{ elementos}}$



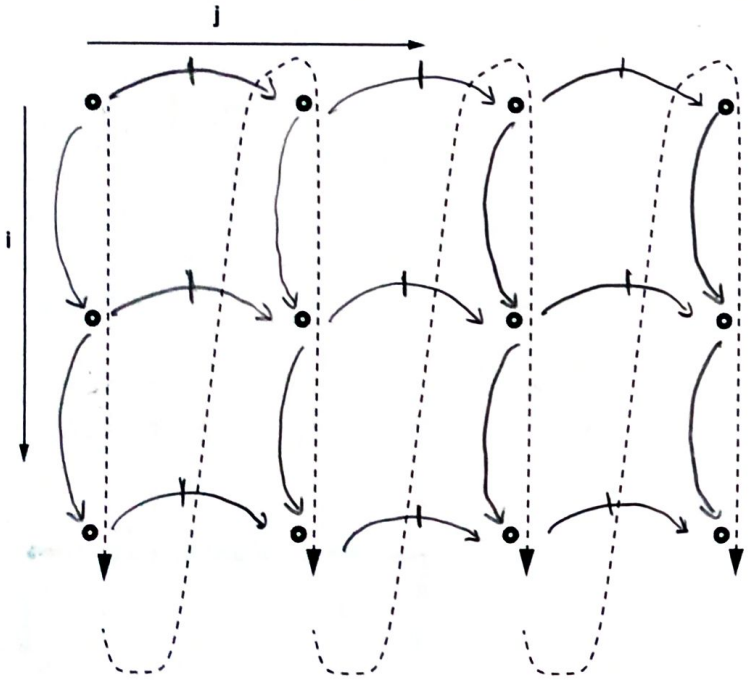
Problema 2

Dependencias, vectorizar y paralelizar **16 puntos**

Dado el siguiente código:

```
real*8 A( , ), B( , )
    ...
do j= 1,N-1
    do i= 1,M-1
S1:     A(i+1,j+1)= A(i,j+1) + B(i,j+1)
S2:     B(i,j)= C(i,j) + 2.0
    enddo
enddo
```

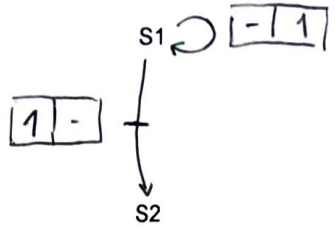
a) Dibuja las dependencias sobre el diagrama de recorrido del espacio de iteraciones:



Queremos ejecutar este código en un multiprocesador de memoria compartida, en el que los procesadores tienen capacidad de ejecución vectorial.

b) Diagrama de dependencias..

dist en j | dist en i



c) Indica si son posibles los modos de ejecución siguientes, añadiendo una explicación muy corta. SEC, VEC y PAR significan ejecución secuencial, vectorial y paralela.

```
j PAR
  i SEC
```
sí/no: no — no se paraleliza j porque hay antidependencias entre iteraciones

```
j SEC
  i VEC
```
sí/no: no — no se vectoriza i porque hay dependencias entre iteraciones

```
j PAR
  i VEC
```
sí/no: no — idem

d) ¿Es correcto intercambiar los bucles? Explicación muy corta.

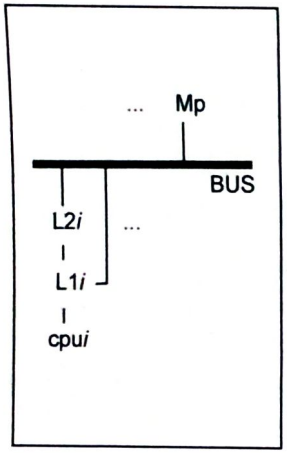
Es correcto ya que mantiene el orden de ejecución del programa.

e) Propón alguna transformación que disminuya el tiempo de ejecución del código original en un multiprocesador vectorial.

```
i SEC
  j VEC
```
Al intercambiar bucles es posible vectorizar el bucle interno, aunque los accesos a memoria son más costosos ya que se producen con stride.

Problema 3

Coherencia de dos niveles exclusivos de cache en bus compartido **35 puntos**



Consideramos un multiprocesador de memoria compartida basado en bus. Cada procesador tiene dos niveles privados de cache cuyos contenidos están en exclusión: $L2 \cap L1 = \emptyset$.

L2, la memoria cache de segundo nivel:

- mantiene coherencia mediante observación e invalidación
- protocolo MSI, con la definición de más abajo
- recibe *todos* los bloques expulsados de L1

L1, la memoria cache de primer nivel:

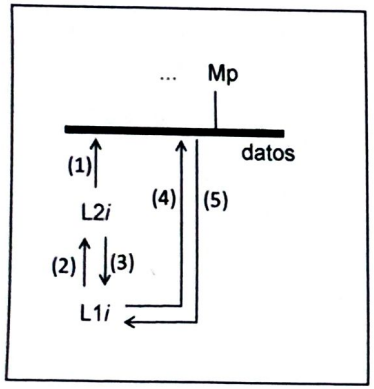
- es de escritura retardada y carga bloque en caso de fallo
- Protocolo MSI convencional, sin observación de bus
- Los comandos que envía a L2, que actúa como intermediaria son: **rB**, **rBinv**, **inv**, **WBclean** y **WBdirty**.

L2 filtra parte del tráfico de coherencia, de forma que L1 únicamente es molestada cuando no hay más remedio.

Vamos a centrarnos en el diseño del protocolo de coherencia del segundo nivel.
La definición de los estados **MSI** de la L2*i* es la siguiente:

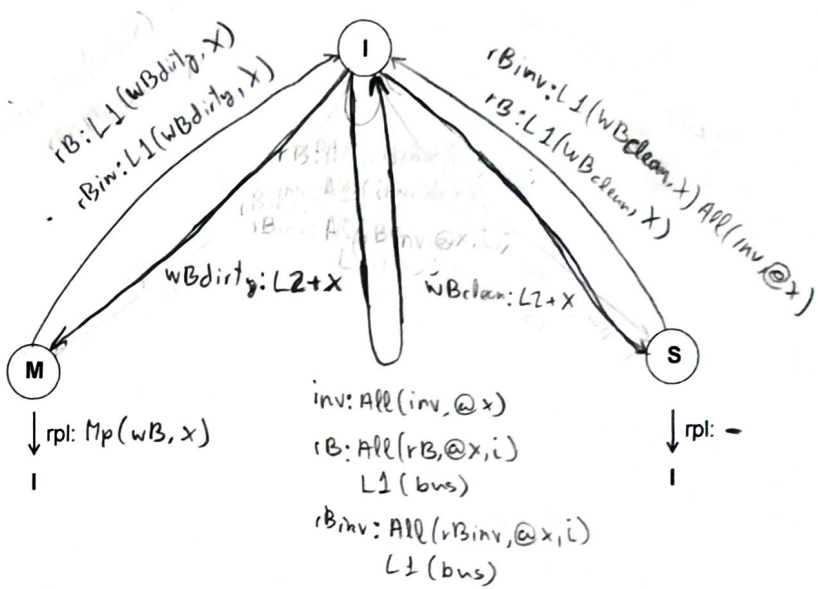
- **I** Como siempre: el bloque *x* no está en el conjunto que toca.
- **M** (1L2 ≠ Mp). Existe una sola copia sucia en todo el sistema, y es ésta.
- **S** (L2*i* = Mp = n(L2*j* ⊕ L1*j*), n ≥ 0, i ≠ j).
  O sea, L2*i* tiene una copia limpia,
  que puede estar replicada, o no, en las caches de otros procesadores.
  Por supuesto, si está en L2*j* no está en L1*j*, y viceversa.

a) Las líneas representan caminos de datos, por donde circulan bloques, las flechas el sentido. Escribe en cada número la condición necesaria para que ocurra ese movimiento de bloque.

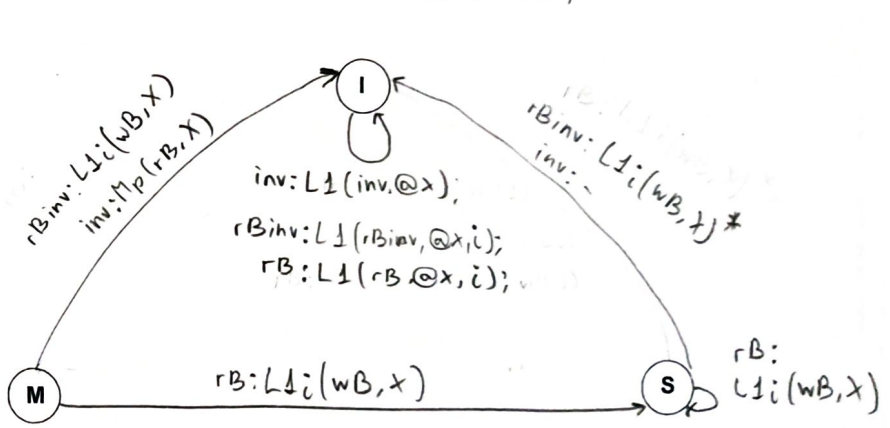


- (1) expulsar bloque sucio a Mp, o mandar bloque de L2i a L1j que lo pide
- (2) expulsar bloque (sucio o limpio) a L2
- (3) L1i pide a L2i un bloque que tiene L2i (rm, wm)
- (4) L1i manda un bloque que le ha pedido L1j.
- (5) L1i recibe un bloque que ha pedido y no estaba en L2i, se lo enviará L2j o L1j.

b) **Protocolo de L2**: transiciones y acciones resultantes de la interacción L1-L2.
Especifica las cargas de bloques desde L1 mediante: **L2 + *x***.



c) **Protocolo de L2**: transiciones y acciones resultantes de la observación del bus.

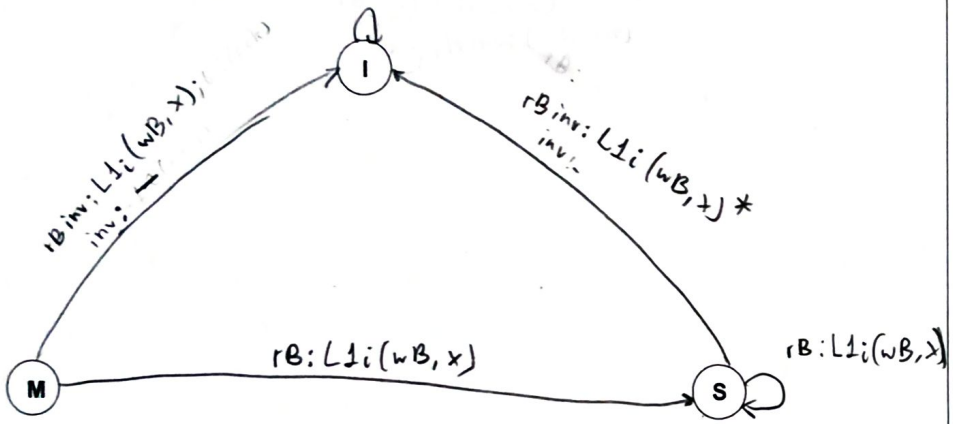


d) L2 puede filtrar los comandos de coherencia que observa en el bus, evitando actividad en su L1, pero no siempre es posible ...
¿Cuándo L2 puede filtrar y cuándo debe molestar a su L1?

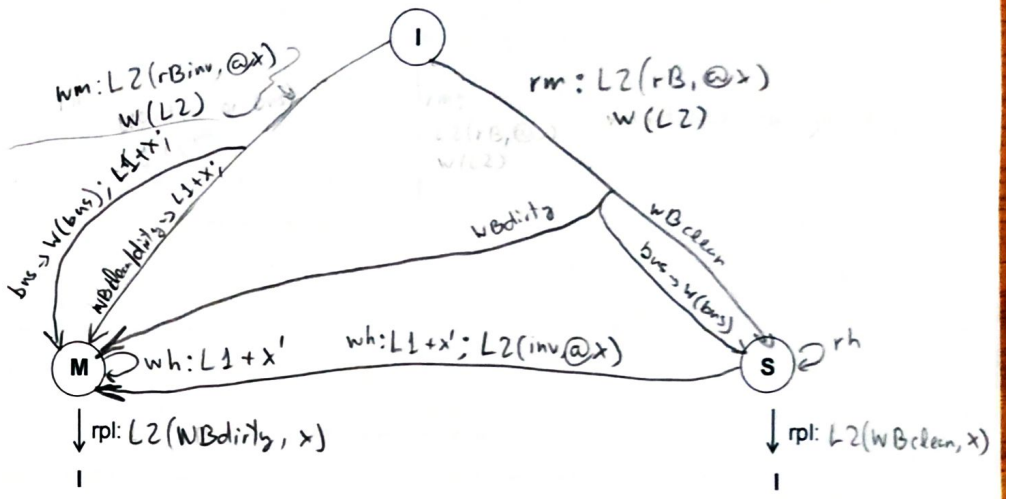
Es necesario molestar cuando L1i quiere un bloque que L2i no tiene.

L1 recibirá el bloque por el bus.

Protocolo de L1. Dibuja las transiciones y acciones en L1 debidas a comandos recibidos desde L2.
Una respuesta negativa desde L1 hacia L2 se representa como: **L2(nack)**.



e) **Protocolo de L1.** Dibuja las transiciones y acciones en L1 debidas a comandos recibidos desde el procesador.



Problema 4

Sincronización **5 puntos**

Escribe en ensamblador (RISC V) el código correspondiente a un semáforo multimarca, es decir, un semáforo que gestiona el acceso concurrente a una región crítica de hasta N hilos de ejecución. Implementa las funciones de `lock()` y `unlock()`. Asume unas primitivas de sincronización tipo load-link y store-conditional.

```
lock(){
        addi  x12, x0, N    // copiamos valor bloqueado
again:  lr.d  x10, (x20)    // load-link a variable L
        beq   x10, x12, again // si no está libre, volvemos a leer
        add   x9, x10, 1    // sumamos semaforo
        sc.d  x11, (x20), x9 // intentamos escribir
        bne   x11, x0, again //si no podemos, volvemos a leer

unlock(){
again:  lr.d  x10, (x20)    //load-link a L
        sub   x9, x10, 1    // restamos semaf
        sc.d  x11, (x20), x9 // intentamos escribir
        bne   x11, x0, again //si no podemos, volvemos a leer.
```


Problema 5

Prácticas Vectorización **14 puntos**

Tenemos el siguiente código C:

```
[...]
for (i = 0; i < LEN; i++) {
    x[i] = alpha*x[i] + beta;
}
[...]
```

Al compilarlo con gcc 7.3, indicando soporte AVX512 con el flag `-mavx512vl`, obtenemos este código ensamblador:

```
.bucke:
    vmulpd (%rax),%zmm3,%zmm0
    add    $0x40,%rax
    vaddpd %zmm2,%zmm0,%zmm0
    vmovapd %zmm0,-0x40(%rax)
    cmp    %rax,%rbx
    jne    bucle
```

a) Responde las siguientes preguntas **con breve justificación**:

a1) ¿Cuántos bytes del vector `x[]` se procesan en una iteración del bucle ensamblador?

AVX512 es una extensión para soportar vectores de 512b = 64 Bytes.

a2) ¿Con qué tipo de dato se está operando (int, float, double ...)?

real → 4 B

a3) ¿Cuántos elementos del vector `x[]` se procesan en una iteración del bucle ensamblador?

$\frac{64\,B}{4}$ = 16 elementos del vector

a4) Sabiendo que al inicio del bucle se realiza la siguiente inicialización:
`%rbx = %rax + 0x4000`
¿cuál es el valor de LEN?

$0x4000 = 16384$

$\frac{16384}{64} = 256$ iteraciones $\therefore$

b) Medimos los siguientes tiempos para una ejecución completa de las versiones escalar y vectorial de este bucle en un Intel Xeon Gold 5120:

- Escalar: 814.2 ns - Vectorial: 188.9 ns

Calcula las siguientes métricas:

b1) Aceleración (*speedup*) de la versión vectorial sobre la escalar.

$Speedup = \frac{814,2}{188,9} = 4,31 = 431\%$

b2) Velocidad de la versión vectorial (GFLOPS).

$\frac{16 \text{ elementos} \cdot 2 \text{ operaciones} \cdot 256 \text{ iteraciones}}{188,9} = 43,36 \text{ GFLOPS}$

Problema 6

Prácticas Paralelización **10 puntos**

Tenemos el siguiente código C:

```
void
gauss_filter(matrix_t * restrict matrix_in, vector_t * restrict vector_in,
vector_t * restrict vector_out){
{
  [...]
  !$OMP PARALLEL DO SCHEDULE(guided)
  for (int i=0; i < width; i++) {
    !$OMP PARALLEL DO SCHEDULE(guided) REDUCTION(+:vector_out[c])
    for (int j=0; j < height; j++) {
      vector_out[i] += (matrix_in[i][j] * vector_in[j]) ;
    }
  }

  [...]
}
```

a) Indica las opciones de compilación **estrictamente necesarias** en cc Sun que paraleliza de forma automática y que muestran en la salida estándar los bucles paralelizados.

`f90 -g -xO3 -autopar -reduction -loopinfo fichero.f90`

b) Añade las directivas OpenMP **estrictamente necesarias** para describir el paralelismo del bucle (sobre el propio código de arriba).

c) Indica las opciones de compilación **estrictamente necesarias** para compilar el código resultante en el apartado anterior.

`f90 -S -xO3 -xopenmp fichero.f90`