Universidad Carlos III de Madrid

Departamento de Informática
Grado en Ingeniería Informática
Arquitectura de Computadores

Examen

ARCOS

ATENCIÓN:

- Lea atentamente todo el enunciado antes de comenzar a contestar.
- **La duración del examen es de 2 horas y 30 minutos.**

NOMBRE:
APELLIDOS:
NIA:

Ejercicio 1 [1 punto]:

Se dispone de una aplicación que permite procesar una imagen de muy alta resolución de la cuál una cierta fracción es paralelizable y otra parte debe ejecutarse secuencialmente. Se asume que no hay límite superior al número de procesos en que se puede paralelizar.

Se desea obtener un *speedup* global de 10 en la versión paralela.

Exprese la fracción de código que debe ser paralelizable en función del grado de paralelismo (número de procesos en que se paraleliza).

SOLUCIÓN:

$S = 1 / [(1-F)+F/n]$

$S \cdot [(1-F)+F/n] = 1$

$S - SF + SF/n = 1$

$nS - nSF + SF = n$

$nS - n = nSF - SF$

$SF(n-1) = n(S-1)$

$F = n(S-1) / S(n-1)$

Para el caso de S=10

$F = n(10-1) / [10(n-1)] = 9n / 10n-10$

Para que tenga sentido F debe ser menor o igual que 1.

$F <= 1$

$9n / 10n-10 <= 1$

9n <= 10n-10

n >=10

Ejercicio 2 [2 puntos]:

Considere un procesador tipo MIPS con arquitectura segmentada (*pipeline*) que tiene dos bancos de registros separados uno para números enteros y otro para números en coma flotante. El banco de registros enteros dispone de 32 registros. El banco de registros de coma flotante dispone de 16 registros de doble precisión ($f0, $f2, …, $f30).

Se supone que se dispone de suficiente ancho de banda de captación y decodificación como para que no se generen detenciones por esta causa y que se puede iniciar la ejecución de una instrucción en cada ciclo, excepto en los casos de detenciones debidas a dependencias de datos.

La siguiente tabla muestra las latencias adicionales de algunas categorías de instrucciones, en caso de que haya dependencia de datos. Si no hay dependencia de datos la latencia no es aplicable.

Instrucción	Latencia adicional	Operación
ldc1	+2	Carga un valor de 64 bits en un registro de coma flotante
sdc1	+2	Almacena un valor de 64 bits en memoria principal
add.d	+4	Suma registros de coma flotante de doble precisión
mul.d	+6	Multiplica registros de coma flotante de doble precisión
addi	+0	Suma un valor a un registro entero
subi	+0	Resta un valor a un registro entero
bnez	+1	Salta si el valor de un registro no es cero

La instrucción bnez usa bifurcación retrasada con una ranura de retraso (delay slot).

En esta máquina se desea ejecutar el siguiente código:

```
bucle:    ldc1 $f0, ($t0)
          ldc1 $f2, ($t1)
          add.d $f4, $f0, $f2
```

```
mul.d $f4, $f4, $f6
sdc1 $f4, ($t2)
addi $t0, $t0, 8
addi $t1, $t1, 8
subi $t3, $t3, 1
bnez $t3, bucle
addi $t2, $t2, 8
```

Inicialmente los valores de los registros son:

- $t0: 0x00100000
- $t1: 0x00140000
- $t2: 0x00180000
- $t3: 0x00000100

Se pide:

a) Enumere las dependencias de datos RAW que hay en el código anterior.
b) Indique todas las detenciones que se producen al ejecutar una iteración del código anterior, e indique el número total de ciclos por iteración.
c) Intente planificar el bucle para reducir el número de detenciones.
d) Desenrolle el bucle de manera que en cada iteración se procesen cuatro posiciones de los arrays y determine el *speedup* conseguido. Utilice nombres de registros reales ($f0, $f2, …, $f30).

IMPORTANTE: Se considerarán incorrectas soluciones que no usen registros realmente existentes (p. ej. $f2’ o $f2’’ son nombres no válidos).

SOLUCIÓN:

a)

```
bucle:  ldc1 $f0, ($t0)        #I1
        ldc1 $f2, ($t1)        #I2
        add.d $f4, $f0, $f2    #I3
        mul.d $f4, $f4, $f6    #I4
        sdc1 $f4, ($t2)        #I5
        addi $t0, $t0, 8       #I6
        addi $t1, $t1, 8       #I7
        subi $t3, $t3, 1       #I8
```

```
bnez $t3, bucle      #I9
addi $t2, $t2, 8     #I10
```

Las dependencias RAW serían:

$f0: I1->I3
$f2: I2->I3
$f4: I3->I4
$f4: I4->I5
$t3: I8->I9

b)

```
ldc1 $f0, ($t0)      #I1
ldc1 $f2, ($t1)      #I2
<stall> x 2
add.d $f4, $f0, $f2  #I3
<stall> x 4
mul.d $f4, $f4, $f6  #I4
<stall> x 6
sdc1 $f4, ($t2)      #I5
addi $t0, $t0, 8     #I6
addi $t1, $t1, 8     #I7
subi $t3, $t3, 1     #I8
bnez $t3, bucle      #I9
addi $t2, $t2, 8     #I10
```

Total: 22 ciclos

c)

```
ldc1 $f0, ($t0)      #I1
ldc1 $f2, ($t1)      #I2
addi $t0, $t0, 8     #I6
addi $t1, $t1, 8     #I7
add.d $f4, $f0, $f2  #I3
subi $t3, $t3, 1     #I8
<stall> x 3
mul.d $f4, $f4, $f6  #I4
```

<stall> x 6
sdc1 $f4, ($t2) #I5
bnez $t3, bucle #I9
addi $t2, $t2, 8 #I10

Total: 19 ciclos

d)

ldc1 $f0, ($t0)
ldc1 $f2, ($t1)
ldc1 $f8, 8($t0)
ldc1 $f10, 8($t1)
ldc1 $f14, 16($t0)
ldc1 $f16, 16($t1)
ldc1 $f20, 24($t0)
ldc1 $f22, 24($t1)
add.d $f4, $f0, $f2
add.d $f12, $f8, $f10
add.d $f18, $f14, $f16
add.d $f24, $f20,$f22
<stall>
mul.d $f4, $f4, $f6
mul.d $f12, $f12, $f6
mul.d $f18, $f18, $f6
mul.d $f24, $f24, $f6
addi $t0, 32
addi $t1, 32
<stall>
sdc1 $f4, ($t2)
sdc1 $f12, 8($t2)
sdc1 $f18, 16($t2)
sdc1 $f24, 24($t2)
subi $t3, 4
bnez $t3, bucle
addi $t2, 32

Total: 27 ciclos cada 4 iteraciones -> 6.75 ciclos por iteración

Ejercicio 3 [1 punto]

Suponga que dispone de un computador con 1 ciclo de reloj por instrucción (CPI) cuando todos los accesos a memoria están en la cache. Los únicos accesos a datos son load y store, y suman el 25% del total de instrucciones. La penalización por fallo es de 50 ciclos de reloj y la tasa de fallo del 5%.

Determine el *speedup* que se obtiene cuando no hay ningún fallo de caché con respecto al caso en que se producen fallos de caché.

SOLUCIÓN:

En el caso sin fallos:

T(cpu) = (ciclos(cpu) + ciclos(espera)) x periodo_reloj = (IC x CPI + 0) x periodo_reloj = IC x periodo_reloj

Al introducir fallos:

Ciclos(espera) = IC x (accesos(instr) + accesos(datos)) x tasa_fallos x penalización_fallo =

IC (1+ 1x0.25) x 0.05 x 50 = IC x 1.25 x 0.05 x 50 = 3.125

T(cpu) = (IC x 1 + IC x 3.125) x periodo_reloj = 4.125 x IC x periodo_reloj

Por tanto el *speedup* será:

S =(4.125 x IC x periodo_reloj) / (IC x periodo_reloj) = 4.125

Ejercicio 4 [1.5 puntos]:

Dada la siguiente definición de una pila libre de cerrojos:

```
template<typename T>
class stack {
private:
        struct node {
                std::shared_ptr<T> data;
                node* next;
                node(T const& data_):data(new T(data_)){}
        };
        std::atomic<node*> head;
public:
        void push(T const& data);
        std::shared_ptr<T> pop()
```

};

a) Proporcione una implementación libre de cerrojos de las funciones push y pop. NOTA: Ignore el problema de los goteos de memoria
b) Explique brevemente como se podría evitar el problema del goteo de memoria asociado a la solución del apartado a.

SOLUCIÓN:

a)

```
template <typename T>
void push(T const& data) {
        node* const new_node=new node{data};
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
}

template <typename T>
std::shared_ptr<T> pop() {
        node* old_head=head.load();
        while(old_head && !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
}
```

b)

Se podría añadir un contador atómico con el número hilos que están realizando un pop y permitir la eliminación cuando no hay hilos en haciendo pop.

Ejercicio 5 [1.5 puntos]:

Dado el siguiente programa que usa OpenMP:

```
double calcula_pi(double step) {
  int i;
  double x, sum = 0.0;
  #pragma omp parallel for reduction(+: sum) private(x)
  for (i=0;i<1000000;++i) {
    x = (i-0.5) * step;
    sum += 2.0 / (1.0 + x*x);
  }
  return step * sum;
}
```

Escriba un programa OpenMP equivalente que no utilice reducciones.

SOLUCIÓN:

```
double calcula_pi(double step) {
  int i;
  double x, sum=0.0;
  int nth = omp_getnumthreads();
  double * sumv = malloc(sizeof(double) * nth);
  for (i=0; i<nth; ++i) sumv[i] = 0.0;

  #pragma omp parallel for private(x)
  for (i=0;i<1000000;++i) {
    int id = omp_getthrednum();
    x = (i-0.5) * step;
    sumv[id] += 2.0 / (1.0 + x*x);
  }
  for (i=0; i<nth; ++i) sum+=sumv[i];
  return step * sum;
}
```

Ejercicio 6 [1 punto]:

Sea un computador con dos procesadores, cada uno con su memoria caché privada que usan política de post-escritura a memoria principal. Inicialmente todas las entradas de las cachés se encuentran marcadas como inválidas. La posición de memoria correspondiente a la variable A, contiene inicialmente el valor 255.

En este computador se ejecuta la siguiente secuencia de operaciones:

Tiempo	Procesador 1	Procesador 2
1	lw $t0, A	
2		lw $t1, A
3	sw $zero, A	
4		lw $t2, A

Indique para cada una de estas operaciones el estado y el valor de la dirección de memoria A tanto en las cachés como en memoria principal

Tiempo	Estado caché 1	Valor caché 1	Estado caché 2	Valor caché 2	Valor Memoria
0	I	--	I	--	255
1					
2					
3					
4					

SOLUCIÓN:

Tiempo	Estado caché 1	Valor caché 1	Estado caché 2	Valor caché 2	Valor Memoria
0	Inválido	--	Inválido	--	255
1	Compartido	255	Inválido	--	255
2	Compartido	255	Compartido	255	255
3	Exclusivo	0	Inválido	255	255
4	Compartido	0	Compartido	0	0