Computación de Altas Prestaciones

Ejercicios de repaso 2023-24

E1.- i) En un problema que solo tiene trabajo serie, W_1, y trabajo con máximo grado de paralelismo, W_n y W'_n, la ecuación de la aceleración, que corresponde a un modelo de memoria limitada (MC), es:

$$S(n) = \frac{W_1 + G(n)W_n}{W_1 + G(n)\frac{W_n}{n}}, \qquad G(n) = 1.5.$$

a) ¿Cómo sería la ecuación de la aceleración para un modelo de carga fija (Ley de Amdahl)? ¿Qué valor de G(n) le corresponde?

$$S(n) = \frac{T(1)}{T(n)} = \frac{W_1 + W_n}{W_1 + \frac{W_n}{n}} = \frac{W_1 + G_1(n)W_n}{W_1 + G_1(n)\frac{W_n}{n}}, \qquad \boldsymbol{G_1(n) = 1}$$

b) ¿Cómo sería la ecuación de la aceleración para un modelo de tiempo fijo (Ley de Gustafson)? ¿Qué valor de G(n) le corresponde?

$$S(n) = \frac{T'(1)}{T'(n)} = \frac{T'(1)}{T(1)} = \frac{W_1 + nW_n}{W_1 + W_n} = \frac{W_1 + G_2(n)W_n}{W_1 + G_2(n)\frac{W_n}{n}}, \qquad \boldsymbol{G_2(n) = n}$$

c) Compare los tres casos anteriores, ordenando la aceleración de menor a mayor y justifique los resultados.

$S_{Amdahl} < S_{\text{memoria limitada}} < S_{Gustfason}$

Si Gn es mayor S es mayor

C1.- ii) En términos generales ¿En qué casos se prefiere una planificación dinámica sobre una estática?

Es preferible el reparto dinámico en casos donde la carga de tareas no es uniforme, para evitar el desbalanceo de carga.

E2) Dado el siguiente lanzamiento de un *kernel* de CUDA,

```
__global__ void kernel(uint8_t* img) {
      uint32_t x = blockDim.x*blockIdx.x+threadIdx.x;
      uint32_t y = blockDim.y*blockIdx.y+threadIdx.y;
      uint32_t w = blockDim.y*dimGrid.y;
      uint32_t nchan = blockDim.z;
      uint32_t c = threadIdx.z;
      img[(y*w + x)*nchan+c] = max(img[[(y*w + x)*nchan+c]*2,255);
}
dim3 A(30, 135, 1);
dim3 B(64, 8, 3);
kernel<<<A,B>>>(img_cuda);
```

¿Cuántos bloques de hilos hay en total?

$$30 \cdot 135 \cdot 1 = 4050\ bloques$$

¿Cuántos hilos por bloque hay? ¿Cuántos warps por bloque hay?

$$64 \cdot 8 \cdot 3 = 1536\ hilos = 48\ warps$$

¿Cuántos hilos hay en total? ¿Cuántos warps hay en total?

$$1536 \cdot 4050 = 6220800\ hilos = 194400\ warps$$

Si el *kernel* procesa un solo elemento de la imagen por cada llamada, ¿cuál es la dimensión de esta? ¿tiene color?

$$dimx = 30 \cdot 64 = 1920$$
$$dimy = 135 \cdot 8 = 1080$$
$$nchan = 1 \cdot 3 \ = 3\ (RGB)$$

Es FullHD y a color con tres canales de color (RGB).

E3) Explique el concepto Pod en Kubernetes.

Pod: unidad mínima operativa en Kubernetes, compuesta de contenedores y almacenamiento. Comparten dirección IP interna del *cluster* los contenedores del mismo Pod.

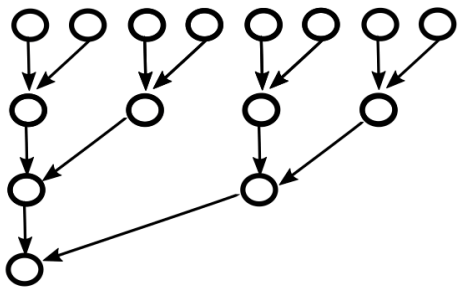
¿Puede un Pod contener varios contenedores?

Sí, puede contener varios contenedores. Esto es útil cuando diferentes contenedores requieren compartir almacenamiento o dirección IP, aunque es habitual el modelo *"One process, one pod"*.

¿Por qué usamos un servicio de tipo NodePort para exponer un puerto al exterior en vez de un servicio de tipo ClusterIP?

Utilizamos NodePort porque este expone el servicio al exterior en el rango de puertos 30000-32767 de todos los nodos del cluster, mientras que ClusterIP solo expone el servicio de manera interna al cluster (i.e. para otros Pods), requiriendo un forwarding manual con `kubectl proxy` si deseáramos acceder al servicio desde fuera.

E4.- En el siguiente grafo se representan las dependencias para m = 2^k-1 tareas. La figura se corresponde con m = 15, k=4 y suponga que cada tarea se ejecuta en una unidad de tiempo.



a) Represente el perfil de paralelismo correspondiente a estas tareas.

b) Calcule el grado de paralelismo máximo y el trabajo realizado con cada uno de los grados de paralelismo que aparecen en el perfil de paralelismo.

DOP max =8 W8 = 8 Δ (suma 0 y envía resultado) , W4 = 4 Δ , W2 = 2 Δ , W1 = Δ

c) Si solo se dispone de la información del perfil de paralelismo, calcule el speedup (S) y la eficiencia (E) para p procesadores y compare los valores concretos de S y de E para p = ∞ y para p =2 S(p=2). ¿ A partir de que número de procesadores no mejora el speedup? ¿Qué eficiencia le corresponde?

S(oo) = S (8) = (8+4+2+1)/ 1+1+1+1= 15/4 = 3,75 |
E(8) = 15/8*4=15/32= 0,46

S(2) = 15/ 4+2+1+1= 15/8 = 1,875 | E(2) =15/16 = 0,9375

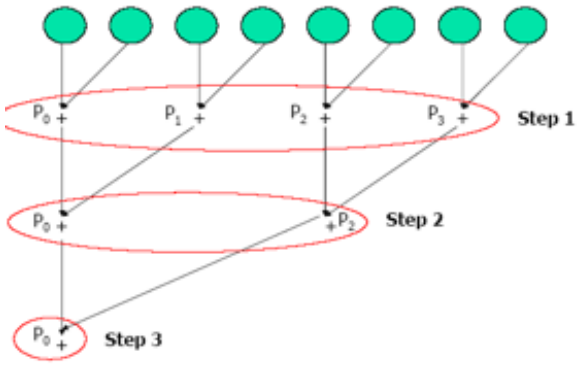
E5.- Se define un sistema con **funcionamiento isoeficiente** si consigue mantener la eficiencia E, constante, conforme se aumenta la carga de trabajo W, y el número p de procesadores.
Suponga que se utiliza la suma de n números para evaluar la eficiencia de un sistema con las siguientes características:

- Cada procesador dispone en memoria local de uno de los números a sumar.
- El programa disponible en cada procesador es una tarea para sumar dos números y por tanto si se desea sumar más de dos hay que repetir tantas veces como sea necesaria dicha tarea.
- El tiempo de cálculo de una suma de dos elementos en un procesador es T
- El tiempo de comunicación para enviar un número a otro procesador es C que supondremos muy pequeño (C<<T).

En estas condiciones el tiempo de referencia en un sistema serie que realice la suma de n números utilizando solo uno de los procesadores es: Tiempo serie = (n-1) x (T+C) y si el tiempo de comunicación es C << T podemos aproximarlo a Tiempo serie = (n-1) x T

Se pide analizar la eficiencia de dos algoritmos que realizan la suma de n números, y que utilizan un grafo de dependencias con una estructura parecida a la analizada en los apartados anteriores, pero teniendo en cuenta que los procesadores en la primera etapa solo envían datos y no realizan suma. La referencia para comparación será el funcionamiento serie descrito anteriormente. Se pide evaluar dos casos:

- Caso n = p (suponga que son potencias de 2) con un algoritmo que realiza los cálculos en etapas, cada etapa con una fase de comunicación y otra de computación, comunicando los datos entre procesadores con una estructura de árbol binario como el de la figura (se representa para n=p=8.



- Caso n >> p suponiendo que cada procesador tiene n/p elementos, pero primero realiza localmente la suma de estos números y con las n/p sumas parciales utiliza un árbol con p procesadores.

Se pide expresar los resultados en función del tamaño del problema, n, el número de procesadores p y calcular el tiempo de ejecución, el Speed-up respecto al algoritmo serie, la eficiencia del sistema y explique en qué condiciones se puede mantener la eficiencia del sistema constante (sistema isoeficiente).
Nota: *puede realizar aproximaciones suponiendo que n es muy grande.*

Solución
Tiempo serie = (n-1) x (T+C) y solo un procesador realiza suma

Caso n = p
Tiempo paralelo = numero de pasos x (T+C) = log p x (T+C) y si C<<T
Tiempo paralelo = log p x T = log n xT
en el paso 1, p/2 procesadores realizan suma y se tienen log p pasos

S = [(n-1)xT]/log n y si n es grande
S = nxT/log n
E = 1/log n no es isoeficiente con la carga de trabajo

Caso n >> p
Tiempo paralelo = tiempos serie suma (n/p) + numero de pasos x (T)
= (n/p -1)xT + log p x T

S = [(n-1)xT]/[(n/p-1)+log p xT y si n es grande
S = n/[n/p + log p] = p/[1 + (pxlog p)/n]

E = S/p=1/ [1+ (pxlog p)/n] si es isoeficiente con la carga de trabajo, por ejemplo si n=pxlog P la eficiencia es constante , E=1/2

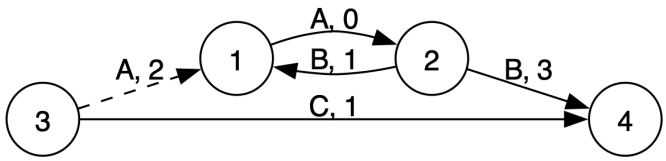
E6.- Dado el siguiente fragmento de código, responda a los siguientes apartados:

```
do i = 2, 1001 // 1000 iteraciones
     (1) A(i) = B(i) + F(i-1) - 10
     (2) B(i+1) = cos(A(i)) + alfa
     (3) C(i-1) = A(i+2) * 3
     (4) E(i) = C(i-2) + B(i-2)
enddo
```

Nota: alfa es un valor constante.

a. Se pide analizar las dependencias entre las instrucciones y generar el grafo de dependencias y el espacio de iteraciones.

RAW de (2) a (1) en B de distancia 1.
RAW de (1) a (2) en A de distancia 0.
WAR de (3) a (1) en A de distancia 2.
RAW de (3) a (4) en C de distancia 1.
RAW de (2) a (4) en B de distancia 3.



b. Escriba el código (pseudocódigo) resultante de la paralelización más eficiente posible sin renombrar variables. Indique claramente los segmentos serie y paralelo del programa.

```
doall i = 2, 1001
     (3) C(i-1) = A(i+2) * 3
enddoall
do i = 2, 1001
     (1) A(i) = B(i) + F(i-1) - 10
     (2) B(i+1) = log(A(i)) + alfa
enddo
 endif
doall i = 2, 1001
     (4) E(i) = C(i-2) + B(i-2)
enddoall
```

No se puede hacer peeling.

c. Indique en el código anterior con `fork` el lanzamiento de hilos, mediante `barrier` la sincronización de hilos, y mediante `join` el proceso de destrucción de hilos. Evite realizar más de un fork y más de un join, ya que estos tardan del orden de diez veces más que un barrier. Evite todos los barrier innecesarios. **Puede usar la variable `thread_id` (como se usa en programaciones tipo OpenMP, MPI o CUDA) para diferenciar el comportamiento de los hilos cuando considere.**

```
fork()
doall i = 2, 1001
     (3) C(i-1) = A(i+2) * 3
enddoall
barrier
if thread_id == 0:
     do i = 2, 1001
          (1) A(i) = B(i) + F(i-1) - 10
          (2) B(i+1) = log(A(i)) + alfa
     enddo
endif
barrier
doall i = 2, 1001
     (4) E(i) = C(i-2) + B(i-2)
enddoall
join()
```

d. Si el tiempo de ejecución de cada instrucción es $T_{inst} = T$, el tiempo de lanzamiento de los hilos es $T_{fork} = 10T$, el tiempo de destrucción de los hilos es $T_{join} = 10T$ y el tiempo de sincronización es $T_{barr} = T$, haga una estimación del tiempo paralelo en función de p, el número procesadores. ¿Cuál es el máximo valor p_{max} del número de procesadores p que hace que el tiempo paralelo sea mínimo? ¿Cuál es el factor de aceleración máximo? ¿y la eficiencia con $p = p_{max}$?

$$T_S = 4 \cdot 1000T = 4000T$$

$$T_p(p) = 2 \cdot 1000T_{inst} + T_{fork} + 2 \cdot \frac{1000}{p} T_{inst} + T_{join} + 2T_{barr} =$$
$$= 2000T + 10T + \frac{2000}{p} T + 10T + 2T$$
$$= 2022T + \frac{2000}{p} T$$

$$S(p = \infty) = S(p = p_{max} := 1000) = \frac{T_s}{T_p(1000)} = \frac{4000}{2024} = 1{,}976285;$$
$$E(1000) = \frac{S(1000)}{1000} = 0.1976304\%$$

E7.- Responda las siguientes preguntas sobre GPU y CUDA.

a) Considere el kernel:

```
__global__ void cuda_thread_start_simple() {
      // const int tid = threadIdx.x + blockIdx.x * blockDim.x;
      const int tid = threadIdx.x; // Código modificado. Original arriba.
      const int elt_per_thread = array_size / num_threads;
      const int start = elt_per_thread * tid;
      const int stop = start + elt_per_thread;
      for ( int h=start; h<stop; h++ ) {
            float4 p = d_v_in[h];
            d_m_out[h] = dot( p, p );
      }
}
```

1. Debido al cambio, el tiempo de ejecución es a) más largo b) aproximadamente igual o c) más corto. Explicar.

2. Debido al cambio, los resultados son a) incorrectos o b) siguen siendo correctos. Explicar.

b) Considere el uso de *__syncthreads()* en CUDA

1. ¿Qué podría salir mal si se eliminara *__syncthreads* del código siguiente?

```
__shared__ int our_data[1025];
our_data[threadIdx.x] = my_element;
__syncthreads();
output_data[tid] = my_element + our_data[threadIdx.x + 1];
```

2. ¿Qué hay de malo en el uso de *__syncthreads* de la manera indicada a continuación?

```
if ( threadIdx.x != 20 ) __syncthreads();
```

c) El ejemplo de código siguiente…

```
__shared__ int sum;
if ( threadIdx.x == 0 ) sum = 0;
__syncthreads();
if ( threadIdx.x == 40 ) sum += 40;
if ( threadIdx.x == 70 ) sum += 70;
if ( threadIdx.x == 200 ) sum += 200;
__syncthreads();
out_data[tid] = sum;
```

1. Indique al menos cuatro valores posibles diferentes para *sum* que el código anterior puede escribir en *out_data*.

2. Explica cómo has obtenido los valores

d) El kernel CUDA A lanzado con 10 bloques de 1024 threads en una GPU con 10 streaming multiprocesos (SMs) tarda 27 s en ejecutarse. Considere una nueva GPU con 9 SM. La única diferencia entre las GPU es la cantidad de SMs. En ambas GPU el número máximo de threads por SM y el número máximo de threads por bloque es 1024.

1. Supongamos que el kernel A se lanza nuevamente con 10 bloques de 1024 threads, pero esta vez en la GPU con 9 SM. ¿Cuánto tiempo tardaría A en ejecutarse en la GPU 9 SM? Explicar.

2. ¿Cuánto tiempo tardaría el kernel A en ejecutarse en la GPU 9 SM si se iniciara con 9 bloques de 1024 threads, pero realizara la misma cantidad de trabajo que el lanzamiento con 10 bloques? El kernel A fue escrito por un programador experto.

Nota: suponga que el tiempo de ejecución es proporcional a la cantidad de trabajo dividida entre los recursos de computación.

SOLUCION

P3. Answer the following questions about GPUs and CUDA.

e) Consider the simple kernel:

```
__global__ void cuda_thread_start_simple() {
      // const int tid = threadIdx.x + blockIdx.x * blockDim.x;
      const int tid = threadIdx.x; // Changed code. Original line above.
      const int elt_per_thread = array_size / num_threads;
      const int start = elt_per_thread * tid;
      const int stop = start + elt_per_thread;
      for ( int h=start; h<stop; h++ ) {
            float4 p = d_v_in[h];
            d_m_out[h] = dot( p, p );
      }
}
```

3. Due to change, execution time is a) longer b) about the same or c) shorter. Explain.

Because of the change every block does the computation that block 0 would do in the original code. Since work is divided evenly, and because the number of blocks hasn't changed, the execution time should not change by much. The only possible reason for an increase in execution time is that each block writes the same array elements. However, the effect of this should be small because the number of writers to a particular location is equal to the number of active blocks (which at most would be on the order of 160 [with a block occupancy of 16 and 10 SMs]) and because for multiple writes to the same location to be a problem those writes would have to occur within cycles of each other, which is unlikely for even two writers because any of a number of factors could cause blocks on different SMs to be at different places in the code such as not starting at exactly the same time.

4. Due to change, results are a) incorrect or b) still correct. Explain.

The results would be incorrect because the computation performed by blocks 1, 2, . . . is not being done.

f) Consider the use of *__syncthreads()* in the CUDA code below( ) and in general.

3. What might go wrong if *__syncthreads* were removed from the code below?

```
__shared__ int our_data[1025];
our_data[threadIdx.x] = my_element;
__syncthreads();
output_data[tid] = my_element + our_data[threadIdx.x + 1];
```

Thread "threadIdx.x" might read element "threadIdx.x + 1" of our_data before thread "threadIdx.x + 1" writes it.

4. What's wrong with the use of *__syncthreads* below?

```
if ( threadIdx.x != 20 ) __syncthreads();
```

All threads in the block must execute syncthreads. In the example above thread 20 does not execute it and so the threads that do execute syncthreads will be stuck in syncthreads.

g) The code sample below...

```
__shared__ int sum;
if ( threadIdx.x == 0 ) sum = 0;
__syncthreads();
if ( threadIdx.x == 40 ) sum += 40;
if ( threadIdx.x == 70 ) sum += 70;
if ( threadIdx.x == 200 ) sum += 200;
__syncthreads();
out_data[tid] = sum;
```

3. Show at least four different possible values for *sum* that the code above can write to *out_data*.

They are: 40, 70, 200, 110, 240, 270, 310.

4. Explain

h) CUDA kernel A launched with 10 blocks of 1024 threads on a GPU with 10 streaming multiprocessors (SMs) takes 27 s to run. Consider a new GPU with 9 SMs. The only difference between the GPUs is the number of SMs. In both GPUs the maximum number of threads per SM and the maximum number of threads per block is 1024.

3. Suppose that kernel A is launched again with 10 blocks of 1024 threads but this time on the GPU with 9 SMs. How long would A take to run on the 9 SM GPU? Explain.

Short Answer: It would take 2 * 27 = 54 s because one SM would be assigned two blocks, and those blocks would run sequentially.
Long Answer: In this particular case, because of the thread limit, only one block at a time can run on an SM. When the kernel is launched each of the first 9 blocks will be sent to an SM. The tenth block will have to wait for one of the first nine blocks to finish. If all blocks take about the same amount of time, that should occur in 27 s. The tenth block will then finish 27 s later, for a total time of 54 s

4. How long would kernel A take to run on the 9 SM GPU if launched with 9 blocks of 1024 threads but doing the same amount of work as the 10-block launch? Kernel A was written by a skilled programmer.

Short Answer: The run time would be 10/9*27 s = 30 s.
Long Answer: Since the programmer is skilled and since the number of blocks matches the number of SMs, we would expect that the run time would be proportional to the amount of work divided by the amount of computing resources. If the amount of computing resources (the number of SMs) changes from 10 to 9 we would expect run time to increase by a factor of 10/9 yielding a run time of 10/9*27 s = 30 s.