Práctica 2: Parámetros de uso de un procesador CMP

Jose Antonio Lorencio Abril

Ejercicio 1

1. El bucle externo no se vectoriza, pues se vectorizan los bucles internos, dentro de lo posible. En algunas ocasiones, puede ser útil usar *outer loop vectorization*, como se explica en `https://software.intel.com/content/www/us/en/develop/articles/outer-loop-vectorization.html`. Claramente, este bucle exterior no es un buen candidato a ser vectorizado, porque la carga de trabajo está, realmente, en los bucles interiores.

   Pasamos a analizar los bucles internos.

   (a) El primero de ellos no puede ser vectorizado, pues contiene dependencias peligrosas. Vemos que B_{i+1} depende de B_i, por lo que la vectorización no puede hacerse. En general, para poder vectorizar este tipo de operaciones, en que los valores de un array dependen de los valores del mismo, debe suceder que se distancien en, al menos, el tamaño del vector. Por ejemplo, en mi caso el procesador puede trabajar con vectores de 256 bits, o sea, 4 doubles, sí podría vectorizarse una operación de la forma $A_i = op\left(A_{i-n}\right),\ n \geq 4$.

   (b) Este segundo bucle sí puede vectorizarse, ya que estamos calculando A_i, dependiente de A_i, B_i y B_{i+1}, dependiente de C_i, de forma que, suponiendo un vector con capacidad para 4 doubles, en cada iteración podríamos hacer el cálculo de $A_j = A_j + B_j$ y de $B_{j+1} = C_i * 2,\ j = i, ..., i+3$. Aunque pueda parecer que A tiene una dependencia, en realidad no es así, pues A_j solo necesita conocer su valor actual para su cálculo, y su valor futuro no afecta a ningún otro elemento.

   (c) En este tercer bucle ocurre como en el primer, A_{i-1} depende del valor de A_i, por lo que no puede ser vectorizado.

   (d) Este cuarto bucle puede vectorizarse de forma bastante clara, usamos los valores de 3 arrays diferentes, no hay dependencias.

   (e) Por último, este último bucle parece tener una clara dependencia, ya que el valor de *total* depende de su valor anterior en toda iteración. No obstante, este caso es muchas veces detectado por los compiladores, que modifican ligeramente el código para que pueda ser vectorizado en alguna medida.

2. Como decíamos, el bucle externo no se vectoriza, pues se vectorizan los internos sin hacer fusión de bucles.

   (a) Como adelantamos, no lo vectoriza, pues encuentra una dependencia entre B_{i+1} y B_i. Para que vectorice, podemos obtener B_{i+4} en función de B_i y los 4 A_i siguientes. Como debemos tener los A_i calculados, separamos el bucle:

```
1  for (int i = 0; i < arrays_size; ++i) A[i] = i* 2/3;
   for (int i = 0; i < arrays_size - 4; ++i) {
         B[i + 4] = A[i+3] - A[i+2] + A[i+1] -
           A[i] + B[i];
       }
```

```
6  for (int i = 0; i < 4; ++i){
       B[i+1] = A[i] - B[i];
     }
```

Sin embargo, esto tampoco soluciona el problema, pues vemos que añadimos, en realidad, muchas más operaciones y la vectorización no es eficiente. Esto se debe a las llamadas dependencias cíclicas, que no tienen solución. Deberíamos cambiar el algoritmo de cálculo.

(b) El segundo bucle es vectorizado, como predijimos. El compilador calcula, en cada loop, $A_i, A_{i+1}, A_{i+2}, A_{i+3}$ y, después, $B_{i+1}, B_{i+2}, B_{i+3}, B_{i+4}$, de forma vectorizada.

(c) El tercero no puede ser vectorizado, debido a la dependencia comentada en el apartado 1, entre A_{i-1} y A_i. Ocurre exactamente lo mismo que en (a), no podemos resolverlo debido a las dependencias cíclicas.

(d) El cuarto se vectoriza, sin sorpresas. Y el compilador hace similar a lo que hacía en b).

(e) La gran sorpresa viene en el quinto bucle. Que se vectoriza a pesar de la dependencia que encontramos. Como decíamos, el compilador está preparado para detectar este tipo de diferencias y hacer modificaciones en el código para vectorizar, al menos, parte del bucle. Una forma que se me ocurre de hacerlo es la siguiente:

```
   double aux[4];
2  for (int i = 0; i < arrays_size/4; ++i) {
           for (int j=0; j<4; j++){
             aux[j] += C[8*i+j];
           }

7  }
   for (int i=arrays_size; i> arrays_size-arrays_size%4; --i) {
           total = total + C[i];
   }
   for (int i=0; i<4; i++){
12   total = total + aux[i];
   }
```

De esta forma vectoriza el bucle interno, y hará los cálculos de 4 en 4.

Una mejora que podemos hacer en este código es sustituir el 4 por un 16, para que sea el array del tamaño del vector multiplicado por la latencia (4 en mi caso), de forma que además de vectorizar, llenamos el cauce de ejecución, mejorando sustancialmente el tiempo del programa.

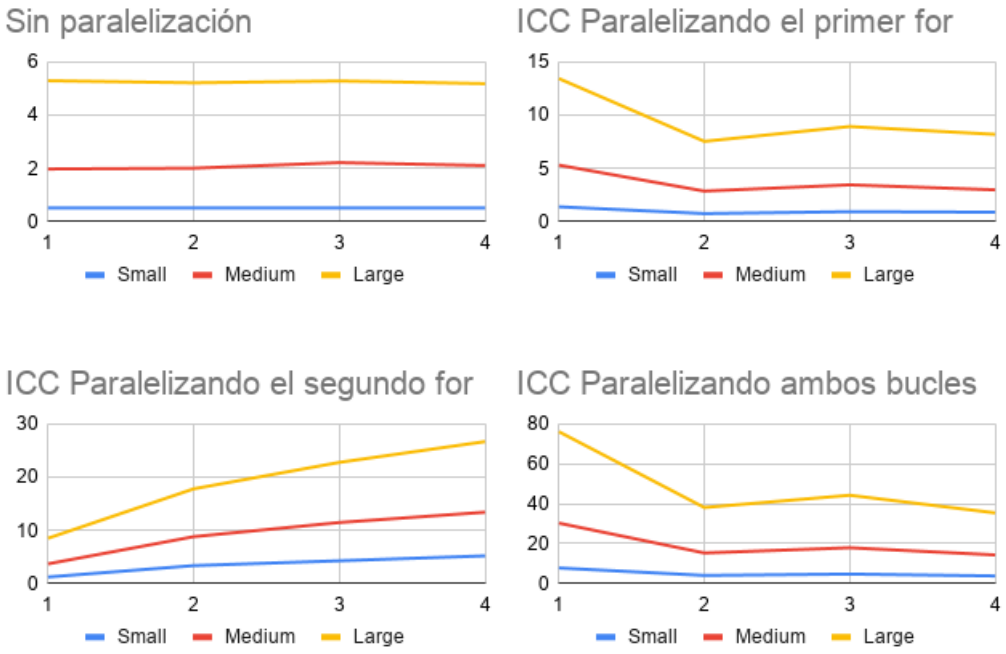
Ejercicio 2

1. El do-while no es paralelizable, ya que cada loop necesita tener la matriz actualizada tras el loop anterior completo.

   Los dos for interiores sí que son paralelizables, pero debemos tener cuidado con la variable difference. Ambos bucles pueden ser paralelizados utilizando el pragma #pragma omp parallel for reduction(max: difference). De esta forma cada hilo calcularía su variable difference y, cuando terminasen, se pondrían los resultados en común utilizando la función max.

   Además, podemos paralelizar los dos for usando la cláusula collapse(2) (#pragma omp parallel for collapse(2) reduction(max: difference)), que junta ambos for en uno solo.

2. He usado compilador de intel instalada en mi máquina. Mi procesador es un intel i5-7200U con arquitectura Kaby Lake y 2.5GHz. El compilador usado es el intel icpc versión 19.1.3.304. Obtenemos las siguientes gráficas al hacer las pruebas:



Estos resultados son bastante sorprendentes. Vemos como nuestra mejor opción sería no paralelizar nada y dejar el programa secuencial. Se está produciendo el llamado **parallel slowdown**, fenómeno en el que, al paralelizar un programa, este va más lento. Una explicación que se me ocurre es que cada hilo no hace casi nada, démonos cuenta de que cada iteración del bucle es hacer una media de 5 elementos y calcular un máximo. Por tanto, entiendo que se produce un efecto cuello de botella en el momento de comunicación de los resultados entre hilos.

Este argumento se respalda, también, al comparar la paralelización del for interior con la del exterior. Cuando paralelizamos el exterior, cada hilo hará todos los cálculos de una fila completa de la matriz, y los resultados se pondrán en común tras terminar toda la fila. Pero al paralelizar el interior, en una misma fila estamos usando los 4 hilos, lo que hace que cada hilo haga menos cuentas y que se necesite más sincronización.

Algo que no consigo comprender es el por qué de la última gráfica. Parece un rendimiento excesivamente malo en comparación con los demás, y no he sido capaz de encontrarlo explicación a este fenómeno.

Por último, comentar como parece haber un mínimo cuando usamos 2 hilos. Esto sucede porque mi ordenador tiene dos núcleos y dos hilos por núcleo. Por tanto, 'prefiere' usar dos núcleos a usar uno, pero también prefiere tener un único hilo por núcleo que dos. Es una especie de tradeoff entre más paralelización y más velocidad por hilo. En nuestro caso el tradeoff se mueve hacia paralelizar lo menos posible.

Comentarios sobre esta práctica

Esta práctica me ha resultado muy interesante, tanto que resulta un poco frustrante no haber podido indagar más en estos temas debido a la carga de trabajo de las últimas semanas.

Con esta práctica hemos aprendido la importancia de intentar programar de forma que el compilador tenga fácil la consecución de optimizaciones, especialmente la vectorización. Esta parte del trabajo me ha parecido que ha quedado muy clara, está bien explicada y los ejercicios bien diseñados para comprender estos conceptos.

Por otro lado, la parte de paralelización nos muestra las dos caras de la moneda. Tenemos un ejemplo en que la paralelización mejora nuestros tiempos de ejecución (el caso de la integral) y otros en los que se produce parallel slowdown, como es el ejercicio que hemos resuelto después. Esto es muy iluminador, ya que te hace darte reflexionar en el porqué no siempre paralelizar es mejor opción, y cuándo, efectivamente, sí que lo es.

Quizás sería provechoso un ejercicio en el que pudiésemos aplicar ambas técnicas y mejorase sustancialmente los tiempos de ejecución, para ver las dificultades que tiene mezclar las técnicas, y sus ventajas.

Como aspecto negativo diría que hay ciertas cosas que debemos usar en la práctica y no están explicadas en el boletín de prácticas, como la directiva 'collapse' para paralelizar bucles anidados o la vectorización de expresiones del tipo $a+ = b$. Entiendo que debemos ser capaces de resolver este tipo de cosas sin que salgan explícitamente en la explicación de la práctica, pero este tipo de cosas básicas creo que hubiera sido mejor saberlas antes, y poder centrarse en otros aspectos más complejos.

Comentarios sobre el artículo de trabajo adicional

El resumen de la historia de la vectorización me hace pensar que la vectorización es realmente la evolución natural de los procesadores. Sobre todo teniendo en cuenta que, como comenta, la introducción de vectores en muchas ocasiones no deriva en un aumento del consumo energético (aunque sí que aumenta la carga del procesador y se debe reducir la frecuencia de reloj) ni de la latencia. Seguro que se producirán avances en este mismo sentido en el futuro.

Por otro lado, vemos claramente como en este artículo se pone de manifiesto que la evolución de la vectorización implica necesariamente un cambio en los compiladores, y ciertamente un cambio muy complejo de implementar de la forma más provechosa. Vemos con los diversos ejemplos realizados, que la mayoría de las veces la vectorización automática no produce mejoras significativas en los programas, precisamente porque el compilador no es capaz de mapear nuestros programas de forma totalmente vectorizada a ensamblador. Para darle un uso completo a esta capacidad, parece que lo que debemos hacer es programar teniéndolo en cuenta, y hacer nuestros programas vectorizados, a posta. Esto, evidentemente, requiere de un esfuerzo muy grande por parte del programador, que probablemente no será asumible en la mayoría de casos.

Al final, tenemos un tradeoff entre la performance de nuestros programas y el tiempo que debemos invertir en su desarrollo. Y, como todo tradeoff, depende de las cantidades relativas entre ambas medidas. Probablemente, se consigan mejorar los compiladores hasta alcanzar mejores rendimientos que los actuales (aunque seguro que no tan buenos como la adaptación propia de cada programa), de forma que haya una ganancia significativa en la ejecución de los programas, sin un aumento excesivo del tiempo de programación. En cambio, el esfuerzo habrá estar en el diseño y mejora del compilador.

Una última reflexión, es que no me parece descabellado que los procesadores sigan evolucionando y se derive en una divergencia mucho más acusada que la actual entre diferentes procesadores, llegando incluso a hacer procesadores dedicados para ciertas tareas específicas. Un ejemplo reciente pero que ilustra lo que quiero decir son las TPUs de Google, circuitos integrados diseñados específicamente para tareas de inteligencia artificial, optimizado para la libreria TensorFlow de la compañía.