Memoria de la práctica de OpenMP

Computación paralela

Manuel de Castro Caballero Pablo Martínez López

1. Inicios

Lo primero que hicimos a la hora de enfrentarnos a la práctica fue seguir las indicaciones de los profesores, por lo que medimos los tiempos de ejecución de las partes del programa (concretamente, de los bloques 3 y 4). Tras realizar la medición, vimos que los bloques de código que más consumían eran el bloque 3, el bloque 4.1, el bloque 4.3 y el bloque 4.8.

Comenzamos intentando paralelizar el bloque 4.1. En este, se rellena la matriz de comida `culture`, generando a partir de una semilla con la función `erand48()` la fila, la columna y la cantidad de comida correspondiente a cada iteración. Tras un par de tardes intentándolo, determinamos que era inviable por varios motivos: las funciones aleatorias, por su naturaleza secuencial, no pueden ser paralelizadas; las llamadas a la función `accessMat()` deben de ser atómicas puesto que cabe la posibilidad de escribir en la misma celda debido a la "aleatoriedad" de la semilla; y paralelizando todo lo posible, los resultados eran ligeramente erróneos en las cifras menos signifiativas.

Pese a no lograr paralelizar el bucle, conseguimos reducir el tiempo de la sección dividiendo los bucles for en dos. El primer bucle realizaría todas las generaciones "aleatorias" y las guardaría en un vector (`VLA`) y el segundo bucle calcularía los valores de `col`, `row` y `food` y modificaría el valor de las posiciones de `culture`. Esto demostró experimentalmente dar mejores resultados que el código original, probablemente debido a una mejor gestión de memoria, por lo que que decidimos mantenerlo.

2. Cuerpo de la práctica

Decidimos dejar de lado el bloque 4.1 para trabajar en el resto de la práctica. Comenzamos aplicando `pragmas` en casi todos los bucles *for* salvo aquellos que vivos desde un principio que no merecían la pena. Modificamos también el inicio del apartado 3 y el apartado 4.2 sustituyendo ambos bucles por la función `memset()`, la cual rellena todas las posiciones de un vector con un valor dado; aunque luego decidimos revertir esta modificación, ya que la función no es paralelizable y los *for* sí. También modificamos todos los bucles que trataban matrices (`culture`, `culture_cells`) para que funcionasen con un bucle *for* en vez de dos, con el fin de hacerlo todo de forma lineal y hacer la paralelización más óptima (similara a hacer un `collapse` "a mano").

Tras realizar las modificaciones y añadir los `pragmas`, no conseguíamos superar el *timewall*, así que intentamos realizar pequeñas modificaciones a ver si finalmente lo conseguíamos. En la sesión de laboratorio de esa semana nos explicaron directivas muy útiles como `schedule`, y tras salir del laboratorio revisamos dónde podríamos aplicar lo recién aprendido. Incluimos las directivas `schedule(guided)` en los bucles 4.3, 4.4, 4.7 y 4.8 pues eran los bucles en que existían casos donde el reparto de tareas no sería óptimo siendo equitativo. En el apartado 3 utilizamos la directiva `schedule(static)` al ser más estable. Incluimos también la directiva de `sections` en los bloques 4.7 y 4.8, puesto que no existen dependencias entre los dos bloques y no debería haber problema al realizarlos simultáneamente.

Al realizar los cambios y lanzar el programa al tablón, los resultados eran erróneos. Estuvimos un día entero buscando sin éxito el problema hasta que un grupo nos comentó que ellos habían entrado en el leaderboard sin haber paralelizado los apartados 4.3 y 4.4, los cuales nosotros si que teníamos paralelizados con directivas `pragma`.

Decidimos reescribir esas secciones con su código original y observamos que los resultados eran correctos. Analizándo el problema, vimos que nos daba error debido a que ambos bloques tienen partes que deben realizarse en exclusión mutua (como las llamadas a `accessMat()` en el 4.3) o variables que necesitan de una `reduction`. Arreglamos los problemas del 4.3 añadiendo reducciones a las variables *step_dead_cells* (utilizando su valor después del bucle para actualizar `num_cells_alive`) y *max_age*, e incluyendo la directiva `pragma omp atomic` a la llamada a `accessMat()` para que se realizase de forma atómica. En el bucle 4.4, utilizamos un `pragma omp critical` para modificar el valor de `step_new_cells` (utilizando aquí también su valor final después del bucle para actualizar `num_cells_alive` y `sim_stat.history_total_cells`) y guardar su valor anterior en una variable auxiliar privada `new`, que poder utilizar como índice de los arrays de las siguientes líneas.

Tras esto, volvimos a lanzar el programa al tablón, superando por primera vez la *timewall*, aunque con un tiempo mucho más alto del que esperábamos. Nos pusimos a analizar el código y vimos que dividir en `sections` los bucles 4.7 y 4.8 no era una buena idea, puesto que el 4.8 tomaba considerablemente más tiempo y dejaba hilos "ociosos" en el 4.7.

Un grupo nos comentó que habían incluido los bloques 4.5 y 4.6 dentro del 4.4, sin paralelizar, y habían obtenido mejora. Pese a que nos sorprendió, decidimos darle una oportunidad, en conjunto a las otras mejoras que habíamos implementado. Cuando volvimos a lanzar el programa al tablón, mejoró nuestro tiempo considerablemente, colocándonos en segunda posición y superando la primera referencia.

3. Buscando mejoras

Tras obtener la segunda posición, intentamos sin demasiado éxito obtener un mejor rendimiento aplicando ciertas mejoras. Principalmente intentamos optimizar el código en general haciendo un mejor uso de la memoria (menos accesos y menos dispersos) y reduciendo el número de iteraciones de los bucles que más tuvieran.

3.1. Pequeñas mejoras

Intentamos realizar pequeñas mejoras a nivel de código para aumentar la optimización, como eliminar variables temporales que se utilizaran poco, hacer *inlining* manual de funciones, mover líneas de código para mantener las variables lo menos dispersas posibles, eliminar *ifs* innecesarios (el primero del bloque 4.3), utilizando `realloc` cuando fuese necesario en vez de `malloc` cada iteración etc... Una de las más notables es que, como las 10 primeras iteraciones de la simulación son distintas (ninguna célula puede morir ni dividirse), las separamos en un primer *for* con código más reducido. También cambiamos los `schedules` de los `parallel for` a base de prueba y error hasta encontrar la configuración óptima. Si estos cambios producen alguna mejora, no es una considerable.

3.2. Reintento de paralelizar el bloque 4.1

Volvimos a intentar paralelizar el bloque 4.1, ya que en un principio observamos que era de los que más tiempo de ejecución añadía. Gracias a las dudas que se respondieron en el canal

de Discord de la asignatura, ahora teníamos una mejor comprensión de los problemas que surgieron anteriormente:

- Nuestro primer bucle es inherentemente secuencial debido a que la generación de números "aleatorios" utiliza una semilla, la cual es una constante del programa, que se modifica en cada llamada a la función. Al paralelizar la generación de datos, algunas llamadas a la función no utilizan el valor de la última semilla, produciendo distintos resultados a los del programa secuencial.
- La matriz de comida, `culture` de tipo `float`. Debido a los límites de la representación de números reales en ordenadores, la suma de `floats` no siempre cumple la propiedad asociativa. Al paralelizar el segundo bucle, los resultados se sumaban a la matriz en orden distinto al secuencial, generando errores en las cifras menos significativas de los resultados.

Intentamos paralelizar tan solo el cálculo de los datos, no su suma, utilizando 3 bucles; pero esto aumentó el tiempo de ejecución del programa. También intentamos paralelizar la generación de comida con la generación de comida en el "sitio especial" (con un `parallel if`), en los casos en los que la segunda procediera, pero los tiempos también empeoraron.

3.3. Intento de mejora de los bloques 4.3 y 4.4

Sabiendo que la paralelización del bloque 4.4 no era beneficiosa, intentamos juntar los bloques 4.3 y 4.4 (junto con los 4.5 y 4.6 ya añadidos al 4.4), ya que ambos realizan el mismo número de iteraciones. Esto resultó ser imposible, ya que cualquier iteración del bloque 4.4 solo puede realizarse con información que solo se conoce cuando el 4.3 termina (notablemente, el número de células en cada celda).

3.4. Intento de reducir el tiempo en el bloque 4.8

Con el fin de realizar menos iteraciones en el bucle 4.8, creamos una estructura auxiliar que contenía las posiciones de `culture` donde existía comida, así como la cantidad de comida. Esto reduciría las iteraciones de los bucles que utilizasen el vector `culture` (`rows` × `columns`) iteraciones. Sin embargo, el tiempo empleado en crear el vector auxiliar hacía que no mereciese la pena dicho cambio.

3.5. Intento de eliminar el bloque 4.2

Como en el bloque 4.2 se realizan muchas iteraciones, lo eliminamos e implementamos su funcionalidad de forma dividida en los bloques 3.2, 4.3 y 4.7 (donde se podía sumar y/o restar 1 de forma atómica a las posiciones de las células según se creaban, movían, dividían, o morían).

La modificación, aunque daba mejores tiempos en nuestras máquinas (con procesador Intel i7 7700K y 16 *cores*), dio peores en el tablón; probablemente porque alguna prueba del tablón presentaba más células que posiciones en la matriz, caso que nosotros no habíamos contemplado.

3.6. Intento de creación de un umbral de paralelización

Como la variable `num_cells` domina las iteraciones de los bucles de varios bloques, notablemente el 4.3 y el 4.4, y su valor cambia con cada iteración de la simulación, consideramos la posibiliad de que, de tener un valor muy bajo, no rentara la paralelización de dichos bucles. Por tanto implementamos un umbral con `parallel if` por el cuál no se paralelizarían ciertos bucles si sus iteraciones no superaban un valor en función del número de hilos utilizados (hallado llamando a la función `omp_get_max_threads()`).

El resultado, de nuevo, aunque podía ofrecer mejoras en nuestras máquinas, no lo hacía en el tablón.

4. Conclusiones

Aunque hayamos quedado quintos y superado la segunda referencia de la *leaderboard*, consideramos que, para el tiempo que le hemos dedicado a la práctica, no hemos obtenido unos resultados tan óptimos como esperábamos. Concluimos que quizás hemos estado tratando el problema como uno secuencial al que había que aplicar cierta paralelización, en vez de pensar en paralelo desde un primer momento. Hemos obtenido una gran optimización paralelizando el programa secuencial, pero con casi con toda seguridad se puede obtener un mayor rendimiento transformando el programa por completo en uno que trabaje realmente de forma paralela.

Hay varios puntos que no hemos podido explorar, ya sea por falta de ideas o por falta de tiempo; como el paralelismo de tareas y no solo de datos o la creación de más estructuras auxiliares para eliminar operaciones secuenciales.

También creemos que con nuestro programa hemos alcanzado un "óptimo local", ya que hiciésemos lo que hiciésemos, por muy buenas que pareciesen las ideas e incluso cuando mejoraban el rendimiento en nuestros ordenadores, nuestro tiempo de ejecución en el tablón aumentaba. Probablemente, y como decimos, sería necesario una aproximación distinta al problema para obtener resultados notablemente mejores.

Intentaremos tener todo esto en mente de cara a las siguientes dos prácticas.

5. Bibliografía

Referencias

[era] erand48() — pseudo-random number generator. https://www.ibm.com/support/knowledgecenter /SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rernd4.htm.

[GS04] Brian Gough and M Stallman. An introduction to gcc for the gnu compilers gcc and g++. 2004.

[mem] C library function - memset(). https://www.tutorialspoint.com/c_standard_library/c_function_memset.htm.