PROGRAMACIÓN PARALELA Y COMPUTACIÓN DE ALTAS PRESTACIONES

TRABAJO AUTÓNOMO: ENTORNOS DE PROGRAMACIÓN PARALELA

PRÁCTICAS DE MPP 17/18

José Luis Cánovas Sánchez

5 OCTUBRE

Práctica 1: Sistemas de programación paralela + OpenMP

CUESTIÓN 1

En `5octubre/cuestion1.cpp` se encuentra el código para multiplicar matrices con paralelismo de C (fork).

Como sólo son 2 procesos para la multiplicación, la división del trabajo se reparte respecto a las filas de la matriz izquierda a.

En el primer comentario indica cómo compilarlo: `g++ -O3 cuestion1.cpp`

Al ejecutar, espera por la entrada estándar los valores:

N = filas de las matrices cuadradas a multiplicar

semilla = semilla para inicializar el generador de números aleatorios

Imprime el tiempo de la multiplicación secuencial, el tiempo en paralelo y la diferencia entre las matrices generadas (si da un valor próximo a cero, las matrices son iguales).

Para N=1000 y semilla=3, tras 10 ejecuciones se obtienen las siguientes medias:

T. secuencial	1055,4 ms
T. paralelo	547,8 ms

La máquina para la ejecución tiene las siguientes características:

Architecture:	x86_64
CPU(s):	8
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
Model name:	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	8192K

CUESTIÓN 2

En `5octubre/cuestion2and3.cpp` está el código para la versión con pthreads.

Se compila con `g++ -O3 cuestion2and3.cpp -fpermissive -lpthread`
Acepta como entrada N y semilla como en cuestion1.cpp, e imprime la misma información.

Para N=1000 y semilla=3, tras 10 ejecuciones se obtienen las siguientes medias:

T. secuencial	1068,9 ms
T. paralelo	534,9 ms

Hay una diferencia de apenas 10ms entre la ejecución en paralelo de C frente a la de Pthreads, siendo Pthreads la más rápida.

CUESTIÓN 3

El código de 5octubre/cuestion2and3.cpp está preparado para cambiar los el número de threads modificando la línea:

```
#define NUM_THREADS 2
```

	T. secuencial	1068,9 ms
2 THREADS	T. paralelo	534,9 ms
4 THREADS	T. paralelo	356,8 ms
6 THREADS	T. paralelo	295,6 ms
8 THREADS	T. paralelo	300,1 ms

A pesar de que con 2 threads se consigue aproximadamente el doble de velocidad respecto a la versión secuencial, la versión con 4 threads es sólo un ~33% más rápida que con 2 threads, y la versión con 6 y 8 threads apenas un ~15% más rápido que con 4 threads. Las versiones con 6 y 8 threads apenas tienen diferencia, siendo además 8 threads un poco más lento. En total, con 6 u 8 threads conseguimos ejecutar en la tercera parte del tiempo que la versión secuencial.

Vemos que utilizando un thread por core obtenemos un buen rendimiento a pesar de paralelizar solo por filas de a, y aprovechando el hyperthreading, podemos mejorar un poco más el rendimiento, siendo conscientes que se trata de varios hilos ejecutándose sobre el mismo core.

CUESTIÓN 4

El código utilizado es el del problema B dado modificando sólo en `sec.c` `NUM_THREADS` y la cláusula schedule del parallel for.

	T. secuencial	1441,7 ms
2 THREADS	T. paralelo	533,3 ms
4 THREADS	T. paralelo	286,8 ms
6 THREADS	T. paralelo	234,0 ms

8 THREADS	T. paralelo	208,6 ms
10 THREADS	T. paralelo	240,9 ms

Los tiempos con OpenMP son parecidos al de versiones anteriores. La ejecución secuencial, que es OpenMP con un hilo, es sin embargo notablemente más lenta que las anteriores ejecuciones, cercanas al segundo de ejecución.

Con 2 hilos de OpenMP conseguimos cerca de 530ms, como antes. Es cuando ampliamos a 4 hilos de OpenMP cuando hay una mejoría de unos 70ms respecto a la versión con Pthreads, del mismo modo con 6 hilos mejoramos en unos ~60ms la versión anterior.

Con 8 threads en OpenMP conseguimos mejores tiempos que con 6 hilos, comparando con lo que ocurrió antes, que obteníamos por poco un peor resultado. Además, con 8 hilos, todos los que puede ejecutar la máquina en paralelo con hyperthreading, nos dan el mejor resultado, mejorando en ~100ms a Pthreads.

Si decidimos usar más hilos de los que pueden ejecutarse realmente en paralelo, vemos cómo los tiempos empeoran.

CUESTIÓN 5

Ejecutando en el concurso de calisto.inf.um la versión OpenMP, sin modificar el schedule, obtenemos (tiempos en ms):

Secuencial			4394
2 Threads			1982
3 Threads			1390
4 Threads			896
2 Threads	Dynamic		2332
3 Threads	Dynamic		1129
4 Threads	Dynamic		1721
2 Threads	Dynamic(2)		2221
4 Threads	Dynamic(2)		812
4 Threads	Dynamic(4)		1063
2 Threads		Collapse(2)	1826
3 Threads		Collapse(2)	2318
4 Threads		Collapse(2)	966
2 Threads	Dynamic	Collapse(2)	2560
4 Threads	Dynamic	Collapse(2)	1841
2 Threads	Dynamic(2)	Collapse(2)	2608
4 Threads	Dynamic(2)	Collapse(2)	2167

Sin opciones añadidas al parallel for obtenemos mejoras equivalentes a las obtenidas en la máquina propia.

El uso de la cláusula schedule(dynamic), el overhead introducido por OpenMP para asignar los trabajos bajo demanda empeora en general los resultados, pero la asignación dinámica mejora si utilizamos schedule(dynamic, 2) para dar chunks de trabajo mayores a los hilos. Si ampliáramos demasiado los chunks de trabajo, podemos volver a peores tiempos si la asignación es peor que la estática, por lo que añadimos el overhead de asignación de trabajo e hilos ociosos sin trabajo.

Al utilizar collapse para unificar los bucles externos que recorren las filas de a y columnas de b, obtenemos tiempos parecidos al de la paralelización del bucle exterior solo, pero en el caso de 3 hilos el tiempo empeora notablemente.

Si combinamos las cláusulas schedule(dynamic) y collapse, por el overhead de un dynamic tan pequeño obtenemos de nuevo tiempos peores que la versión sin cláusulas de parallel for. Al ampliar a chunks mayores obtenemos tiempos con 4 hilos incluso peores que con 2 hilos sin cláusulas.

Podemos concluir que el uso de cláusulas OpenMP pueden ser beneficiosas utilizadas junto a un código que pueda dar el máximo provecho. El código de multiplicación de matrices utilizado sólo tiene en cuenta la memoria en cuanto a las variables privadas i, j y k. Si la división de trabajo de OpenMP hace que hilos distintos con memorias caché compartidas lean zonas de la matriz lejanas, los fallos de caché por reescribirse unos a otros pueden hacer más lento el programa. Un ejemplo de mejora de uso de la memoria, para cualquier versión paralela de multiplicación de matrices, sería aplicar la traspuesta a b, y multiplicar recorriendo las filas de a y b^t (columnas de b), de modo que si una matriz se guarda en memoria por filas, la localización temporal se aprovecha en la caché de memoria.

26 OCTUBRE

Práctica 2: Programación híbrida. MPI y MPI+OpenMP

CUESTIÓN 1

En `26octubre/D/` se encuentra `sec1.c` la versión con `MPI_Gather()` que llaman todos los procesos de la misma forma, eliminando los `if else` que comprueban el nodo.

En Mooshak, con la cuenta `CanovasSanchez17` el envío 527 (6244ms) corresponde al código base, y el envío 528 (6465ms) corresponde a la versión con Gather.

CUESTIÓN 2

En `26octubre/D/` se encuentra `sec2.c` la versión con `MPI_Pack()` donde primero se empaqueta la matriz b y se guarda la posición en el buffer, de modo que al empaquetar la parte distinta de la matriz a correspondiente a cada proceso, no hay que volver a empaquetar la matriz b.

El envío 529 (Runtime Error) corresponde al un error al no poner correctamente el tamaño del buffer, que debe ser en bytes y no en cuántos double contiene. El envío 530 (6385ms) corrige esto usando un buffer como array de char, y pasando a MPI_Pack el tamaño correcto.

CUESTIÓN 3

En `26octubre/D/` se encuentra `sec3.c` la versión con MPI+OpenMP. Los envíos del 531 al 538 son las otras versiones con MPI+OpenMP. La implementación es sencilla, añadiendo en la función `mms()` antes del `for` las líneas:

```
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(i, j, k, s) schedule(dynamic)
```

Además de añadir `#define NUM_THREADS 4` al inicio del fichero.

Esta es la versión de MPI+OpenMP que mejor tiempo ha obtenido en Mooshak de entre todas las combinaciones de número de hilos y cláusulas de omp parallel. El envío 538 (4985ms) consigue bajar

de los 5 segundos, y le siguen de cerca las versiones con 6 hilos que usan dynamic, envío 537 (5065ms), y sin dynamic, envío 533 (5001ms).

CUESTIÓN 4

En `26octubre/BC/` se encuentran `mpi.c` y `mpi-openmp.c`, correspondientes a los ejercicios B y C de Mooshak. En el envío 260 (24728ms) obtenemos el tiempo de la versión secuencial. El envío 261 (15282ms) es la versión MPI con 2 procesos. Los envíos 271 (9514ms) y 272 (8940ms) corresponden a las versiones MPI+OpenMP, la segunda añadiendo `schedule(dynamic)`.

Para la solución MPI utilizo tanto `MPI_Scatter()` para repartir la matriz `a` y el array `zerosrows` en `n/np` partes, como MPI_Gather() para recibir la solución `c`. El problema viene cuando la dimensión de la matriz es divisible por el número de procesos. El envío a Mooshak utiliza solo 2 nodos pues todas las pruebas son divisibles por 2, pero cuando se utilizan 4, con este código falla. Una solución sería repartir chunks de tamaño `n/np` excepto para un nodo que recibiría `n%np` filas a multiplicar. He intentado varios envíos pero como en mi máquina tengo que reparar MPI por unos errores que no ocurren en calisto, no he podido arreglar el código.

Para la solución con OpenMP + MPI, parto de la solución MPI válida de antes y lanzo hilos OpenMP en cada nodo para paralelizar la multiplicación de matrices en el bucle for más externo. Compruebo que al añadir `schedule(dynamic)` como antes, se mejoran el tiempo.

9 NOVIEMBRE

Práctica 3: CUDA y XeonPhi

CUESTIÓN 1: CUDA EN MOOSHAK

En el código de Francisco Muñoz cambiamos los tamaños de bloque de 16x16 de la función `sec()` por los mostrados en la tabla con sus tiempos (ms):

4x4	8x8	12x12	16x16	24x24	32x32
1286	947	1466	1718	2501	3274

En la máquina saturno obtenemos que el mínimo se obtiene alrededor del tamaño de bloque 8x8, que en esta máquina concreta debe ser el mapeo lógico que mejor se adapta a la estructura real física de la GPU.

Los envíos en Mooshak corresponden del 925 al 930.

CUESTIÓN 2: CUDA EN HETEROSOLAR

En la cuenta y directorio `mpp17-45/algmatpar` se encuentran los ficheros del código, script pbs y resultados de las ejecuciones en Saturno y Júpiter. La entrada para las pruebas (basada en `esquema-cuda.cu` de Mooshak) consiste en 4 matrices de tamaños entre 1800 y 3000, con máscaras de tamaño entre 48 y 50.

Los tiempos (ms) para los mismos tamaños de bloque de antes:

Saturno:

4x4	8x8	12x12	16x16	24x24	32x32
5525	4081	6336	7395	10765	14108

Júpiter:

4x4	8x8	12x12	16x16	24x24	32x32
3850	2157	4184	4611	5386	6720

Vemos que en Saturno el mínimo sigue situándose en bloques de 8x8, y lo mismo ocurre en Júpiter.

CUESTIÓN 3

En `9noviembre/sec3.cpp` se encuentra la versión con offload al Xeon Phi de Venus, correspondiente al envío 938, con 676ms. En el envío 931 (679ms) se utiliza la función `count()` con `__attribute__((target(mic)))` para llamarlo desde código corriendo en el propio Xeon (sección offload de `sec()`). En la versión del envío 938 implementamos la función count dentro de sec, de cara a la vectorizaicón con simd de la cuestión 4.

En ambos envíos se paraleliza en el bucle for más externo, asignando 224 threads a la tarea (corresponden a los 4 threads por core de los 56 cores libres del Xeon Phi, pues el 57 se encarga de la comunicación con la CPU).

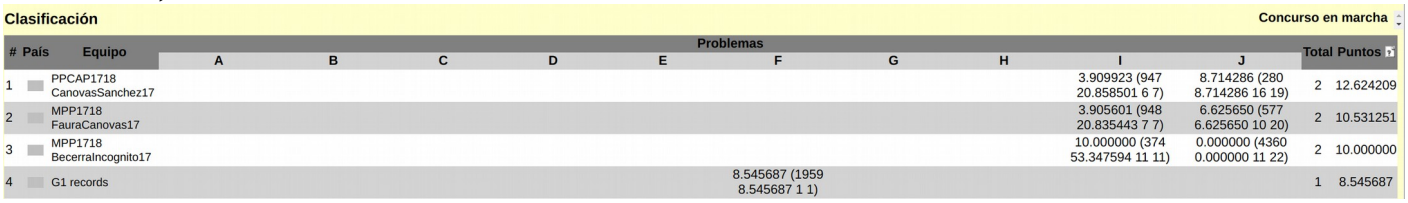
CUESTIÓN 4

En `9noviembre/sec4.cpp` se encuentra la versión con la cláusula simd, correspondiente al envío 949 (280ms), cerca de 2.5 veces más rápido que la versión anterior. En este caso se añade la opción simd en el segundo bucle for. De este modo repartimos el primer bucle for con omp parallel entre los 224 threads del Xeon Phi, y cada thread vectorizará el segundo bucle.

Comparando con el envío 944 (1017ms), donde la cláusula simd se pone en el tercer for (inicio de la antigua función `count()`), obtenemos tiempos algo mayores que en la cuestión 3: el overhead de vectorizar supera al paralelismo obtenido.

En conclusión, paralelizando en los bucles más externos dividimos mayores cargas de trabajo y obtenemos mejores resultados en general.

Finalmente, vemos que el código con bloques de 8x8 de la cuestión 1 y la versión offload con simd de la cuestión 4 obtienen de los mejores tiempos en la clasificación de los problemas I y J del concurso Mooshak, a fecha 12 de noviembre:



Clasificación — Concurso en marcha

#	País	Equipo	Problemas A	B	C	D	E	F	G	H	I	J	Total	Puntos
1		PPCAP1718 CanovasSanchez17									3.909923 (947 20.858501 6 7)	8.714286 (280 8.714286 16 19)	2	12.624209
2		MPP1718 FauraCanovas17									3.905601 (948 20.835443 7 7)	6.625650 (577 6.625650 10 20)	2	10.531251
3		MPP1718 BecerraIncognito17									10.000000 (374 53.347594 11 11)	0.000000 (4360 0.000000 11 22)	2	10.000000
4		G1 records						8.545687 (1959 8.545687 1 1)					1	8.545687