PROGRAMACIÓN PARALELA Y COMPUTACIÓN DE ALTAS PRESTACIONES

TRABAJO AUTÓNOMO: ENTORNOS DE PROGRAMACIÓN PARALELA

PRÁCTICAS DE PPCAP 17/18

José Luis Cánovas Sánchez

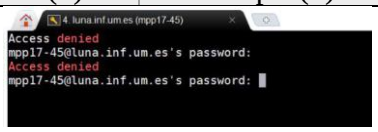
ALGORITMOS MATRICIALES PARALELOS

CUESTIÓN 2

Comparar las prestaciones de una multiplicación de matrices según el bucle de los tres que se paralelice, con distintos tipos de reparto (static o dynamic) y tamaño de división del trabajo, efecto de collapse.

En el directorio `2/` adjunto, se encuentra el código fuente de la paralelización. Debido a que a la hora de hacer el ejercicio luna me denegaba el acceso por SSH (y a Adrián Javaloy tampoco, por lo que debía ser un problema general), tomo los tiempos (ms) en Mooshak de Calisto, problema B:

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



En 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 más los 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.

CUESTIÓN 4

Incluir paralelización de bucles (en los bucles más externos) en una versión de la multiplicación de matrices por bloques y comparar los tiempos de ejecución con la versión por bloques no paralela y con la versión paralela en la que se paraleliza el trabajo dentro de cada bloque.

En el directorio `4/` adjunto, o en `mpp17-45/algmatpar/4/` de luna, se encuentra el código fuente de las versiones paralelizadas en los bucles externos e internos. Comparamos con la tabla de tiempos (segundos) del código secuencial por bloques ejecutado en marte:

Algoritmo	N=500	N=1000	N=2000	N=3000	N=4000
Secuencial					
Tamaño de bloque: 25	**0.177865**	1.453277	11.751641	39.685550	93.973494
Tamaño de bloque: 50	0.180716	**1.361554**	**10.562337**	**35.292006**	**83.937251**
Tamaño de bloque: 100	0.188849	1.376489	10.777013	36.056393	84.532981
Paralelización bucle interno					
Tamaño de bloque: 25	0.074525	0.567204	4.449876	14.790170	35.241169
Tamaño de bloque: 50	0.053506	0.350413	2.735158	9.259358	21.996043
Tamaño de bloque: 100	**0.047331**	**0.315868**	**2.395529**	**8.036172**	**18.861149**
Paralelización bucle externo					
Tamaño de bloque: 25	**0.049491**	**0.303321**	2.387386	7.748638	18.723906
Tamaño de bloque: 50	0.051893	0.329063	**2.177237**	**6.901120**	**16.996734**
Tamaño de bloque: 100	0.054400	0.328252	2.454313	7.003805	17.380692

Se utiliza la función `omp_get_max_num_threads()` para asignar el máximo número de threads admitidos por marte. El detalle más importante a tener en cuenta es reservar un array de matrices s, donde se acumularán las multiplicaciones de matrices con suma dentro de cada hilo, para que no interfieran entre sí. Vemos que en todos los casos mejoramos más del doble al mejor tiempo secuencial.

Entre las versiones paralelizadas, a mayor tamaño de las matrices se observa la ventaja de paralelizar los bucles externos, donde repartiendo más trabajo de cómputo entre los hilos aprovechamos la división en bloques en las múltiples cachés de los cores, así como evitar el overhead de fork-join de hilos en cada multiplicación.

CUESTIÓN 9

Programar una multiplicación de matrices con MPI+OpenMP, y comparar los tiempos de ejecución variando el número de procesos y threads, incluyendo el caso de un único proceso (solo programación OpenMP) y de un solo hilo por proceso (solo paralelismo MPI).

En el directorio `9/` adjunto, o en `mpp17-45/algmatpar/9/` de luna, se encuentra el código fuente de la paralelización con OpenMP + MPI. Para decidir el número de nodos MPI, se modifica la línea del script PBS donde se indican los nodos a ejecutar. Para indicar cuántos hilos OpenMP lanzar, la función main lee por la entrada estándar, en orden, el tamaño de las matrices, el número de hilos OpenMP, y los tamaños inferior y superior de los valores aleatorios a generar.

La ejecución puramente secuencial se realiza en Jupiter. La ejecución con sólo paralelismo de OpenMP se realiza lanzando 24 hilos en Jupiter, el máximo que admite (12 cores con hyperthreading). La versión con solo paralelismo MPI utiliza los 12 cores de Jupiter y los 12 cores de Venus, en teoría idénticos. Finalmente, se combina OpenMP con MPI lanzando en los 24 cores de Jupiter y Venus, con 2 hilos de OpenMP cada uno (para aprovechar el hyperthreading).

Los tiempos en segundos son los siguientes:

Nº nodos MPI	Nº hilos OMP	Tamaño matrices 5000	Tamaño matrices 6000
1	1	503.092819	780.130558
1	24	44.825152	82.421861
24	1	44.277304	70.513559
24	2	**42.431160**	**67.348325**

Vemos que los tiempos con 24 hilos y 24 nodos son muy parecidos, pero a pesar de haber tiempo de paso de mensajes entre 24 nodos en 2 máquinas distintas, se obtiene un mejor tiempo con 24 nodos frente al uso del mismo tamaño de paralelismo usando 12 cores con hypertrheading.

Al aprovechar el hyperthreading con MPI mejoramos un poco los tiempos, pero ni de lejos a la mitad. Esto nos indica que el hyperthreading es una herramienta útil, pero que efectivamente no puede llegar a funcionar como si fueran dos hilos ejecutándose en cores individuales.

CUESTIÓN 14

Comparar el comportamiento de distintas versiones de la multiplicación de matrices con OpenMP en forma offload en venus. Experimentar con schedule, collapse y simd los tiempos de las multiplicaciones matriciales.

En el directorio `14/` adjunto, o en `mpp17-45/algmatpar/14/` de luna, se encuentra el código fuente de la multiplicación de matrices paralela utilizando la transpuesta de b. Se descarga el trabajo de la multiplicación al Xeon Phi de Venus con la línea:

```
#pragma offload target(mic) in(a:length(fa*ca)) in(bt:length(fb*cb)) inout(c:length(fc*cc))
```

Dentro del coprocesador, paralelizamos con OpenMP, probando con distintas cláusulas. Los resultados (en segundos) se muestran abajo:

Tamaño de Matriz =	2000	3000	4000	5000	6000	7000
Sin cláusulas extra	2.921431	4.335710	**6.903049**	12.213944	19.344727	30.088533
Static	2.917755	4.537175	6.978585	12.350229	18.973087	29.916222
Static 5	2.977286	4.357692	7.292801	12.337511	19.444749	29.858921
Dynamic	2.939719	4.299595	7.069046	11.723230	18.649555	**29.300287**
Collapse	**2.744677**	**3.840926**	6.963857	**11.362626**	**18.400358**	29.407477
Simd	3.594959	6.200148	17.879717	20.294112	30.960325	70.273822
Simd 2	2.897114	4.431727	6.941834	12.246327	19.320921	30.047152

En la tabla de arriba, simd y simd 2 hacen referencia al bucle donde se añade la vectorización. El primero corresponde al bucle que recorre las columnas de b (filas de bt), y simd 2 hace referencia al bucle del índice k donde se hacen las operaciones de multiplicación y suma.

Los mejores tiempos los obtiene en general la versión que utiliza la cláusula collapse, seguido de cerca del resto de versiones.