Secuencial

El algoritmo secuencial es muy similar al de la entrega 1. Primero se hace el chequeo de parámetros, la alocación y posterior inicialización de las matrices que correspondan.

La primera decisión que tomamos es a la hora de calcular R1, R2 y sus respectivos promedios.

Decidimos calcular R1 y R2 en un solo recorrido de ambos, y aprovechar la estructura de control para calcular la sumatoria de las mismas.

Luego, calculamos el promedio de cada matriz, y guardamos la multiplicación en una sola variable para su posterior uso en el cálculo de C.

Lo que le sigue es la multiplicación de la matrices, que como vimos en la entrega 1, utilizando multiplicación por bloques es la mejor forma de realizarla. En particular usamos un tamaño de bloque de 64 (el óptimo, demostrado en la entrega 1) y organizamos A y B por columnas.

Por último queda el cálculo de C, el cual se puede hacer todo junto o separar ((R1A[i*N+j] + R2B[i*N+j]) * promedioR1R2).

Tiempo de ejecución:

N (bs = 64)	Versión secuencial
512	0.561895
1024	4.352408
2048	34.835465
4096	269.095233

Pthread

Pthread nos permite hacer uso de los hilos, lo cual nos va a permitir paralelizar el algoritmo y obtener mejores tiempos de ejecución.

La primera sección de código que se ve afectada es el cálculo de R1, R2 y sus respectivas sumatorias. A diferencia de en la versión secuencial, cada hilo va a resolver un conjunto de iteraciones, en nuestro caso el conjunto de iteraciones se separa en partes iguales para cada hilo, ya que son independientes entre sí, dividiendo la matriz en los hilos.

En particular, decidimos dejar junto el cálculo de R1 y R2, ya que fue la forma que demostró mayor eficiencia.

Otra alternativa interesante que probamos fue separar la sumatoria de R1 y R2 de su inicialización, pero tampoco fue una decisión que incrementó la eficiencia, por lo que dejamos todo junto.

Una vez terminada la inicialización de R1 y R2, con el cálculo de las sumatorias respectivo, los hilos se encuentran en la barrera “avrgBarrierMaster”, la cual no solo cordina a los hilos para que se esperen antes de empezar a multiplicar las matrices (ya que antes necesitan que se termine de inicializar R1 y R2), sino que también “habilita” al hilo master para empezar a calcular el promedio (ya que debe esperar al cálculo de la sumatoria).

El cálculo del promedio es una sección secuencial, por lo que no podemos sacar provecho del paralelismo en esa acción.

Luego, realizamos la multiplicación de matrices (R1 con A y R2 con B) por bloques, en este caso también hacemos uso del paralelismo, aplicamos la misma idea que utilizamos al inicializar R1 y R2, la de dividir la matriz en partes, donde cada hilo se encarga de hacer una sección de la multiplicación.

Después de la multiplicación, todos los hilos se quedan esperando en una barrera compartida de nuevo con el master, a la cual el master llega cuando termina de calcular el promedio, de esta manera los procesos esperan a tener el cálculo del promedio listo antes de calcular el resultado de C.

Por último, al pasar la barrera, los hilos se vuelven a repartir la matriz C, dividiendo como en los casos anteriores las iteraciones del doble-for. De esta manera también aprovechamos el paralelismo en el cálculo total de la ecuación.

Para esta última parte, el cálculo de C, también probamos juntando y separando sus terminos en diferentes iteraciones, calculando (R1A+R2B)*avgR1*avgR2 por un lado y posterior suma con T en otro bucle, pero como era de esperar, por las pruebas hechas en la entrega anterior, es mejor la resolución del cálculo en un mismo doble-for, que así fue como lo dejamos.

Tiempo de ejecución:

N (bs = 64)	Versión Pthread		
	T = 2	T = 4	T = 8
512	0.291988	0.147274	0.075757
1024	2.230519	1.126372	0.578590
2048	17.617868	8.898398	4.521590
4096	135.592896	68.233959	34.857703

OpenMP

En el caso del OpenMP, es muy similar a con Pthread, las partes paralelizables son las mismas y la parte secuencial también lo es. Lo paralelizable lo hacemos con **#**pragma omp for, en particular el for que calcula la sumatoria también hace uso de la sentencia reduction con la operación de suma, para que al terminar la ejecución se sumen las sumatorias parciales generadas por cada hilo. Para la parte secuencial, el cálculo del promedio, usamos la sentencia **#**pragma omp single, para que sea ejecutada por un solo hilo. La sentencia single viene acompañada de nowait, ya que los demás hilos pueden empezar con la multiplicación por más que uno esté calculando el promedio.

Como es evidente, todo el código es mucho más legible y fue mucho más fácil de programar por las facilidades que ofrece la librería.

Tiempo de ejecución:

N (bs = 64)	Versión OpenMP		
	T = 2	T = 4	T = 8
512	0.292288	0.147484	0.075324
1024	2.233557	1.126982	0.578433
2048	17.620263	8.928093	4.526614
4096	135.918132	68.412556	34.955603

Validación de los resultados

Para validar que el resultado sea el correcto en todas las soluciones, tomamos un set de pruebas:

Datos de las matrices fuente:

```
M[0*N+0] = 1.849267;
M[0*N+1] = 1.880657;
M[0*N+2] = 2.602202;
M[0*N+3] = 3.200894;
M[1*N+0] = 1.322647;
M[1*N+1] = 1.695682;
M[1*N+2] = 2.207362;
M[1*N+3] = 1.885152;
M[2*N+0] = 0.369521;
M[2*N+1] = 5.444304;
M[2*N+2] = 1.489501;
M[2*N+3] = 2.788930;
M[3*N+0] = 4.134537;
M[3*N+1] = 3.693435;
M[3*N+2] = 5.419336;
M[3*N+3] = 0.957360;

A[0+N*0] = 44.919447;
A[0+N*1] = 98.381286;
A[0+N*2] = 11.605420;
```

```
    A[0+N*3] = 23.060870;
    A[1+N*0] = 22.533610;
    A[1+N*1] = 25.007298;
    A[1+N*2] = 59.790682;
    A[1+N*3] = 61.902849;
    A[2+N*0] = 88.401016;
    A[2+N*1] = 13.863665;
    A[2+N*2] = 22.882454;
    A[2+N*3] = 44.611562;
    A[3+N*0] = 35.187721;
    A[3+N*1] = 27.598218;
    A[3+N*2] = 54.181643;
    A[3+N*3] = 27.377421;

    B[0+N*0] = 6.154822;
    B[0+N*1] = 98.238501;
    B[0+N*2] = 83.896403;
    B[0+N*3] = 69.790850;
    B[1+N*0] = 96.628351;
    B[1+N*1] = 54.513603;
    B[1+N*2] = 10.466182;
    B[1+N*3] = 83.246194;
    B[2+N*0] = 15.625164;
    B[2+N*1] = 38.986050;
    B[2+N*2] = 74.820711;
    B[2+N*3] = 38.559370;
    B[3+N*0] = 73.084614;
    B[3+N*1] = 44.715461;
    B[3+N*2] = 44.131227;
    B[3+N*3] = 75.932759;
```

```
    T[0*N+0] = 55.784399;
    T[0*N+1] = 3.531480;
    T[0*N+2] = 47.037230;
    T[0*N+3] = 63.655909;
    T[1*N+0] = 8.301220;
    T[1*N+1] = 51.852553;
    T[1*N+2] = 50.156271;
    T[1*N+3] = 59.840766;
    T[2*N+0] = 36.454570;
    T[2*N+1] = 28.783482;
    T[2*N+2] = 74.903460;
    T[2*N+3] = 65.783394;
    T[3*N+0] = 92.862907;
    T[3*N+1] = 93.974956;
    T[3*N+2] = 17.091994;
    T[3*N+3] = 21.906788;
```

Resultados de la matriz C:

Para el algoritmo secuencial:

Matriz C

[0,0] = -58792766.396836
[0,1] = -44083996.978909
[0,2] = -56297514.088618
[0,3] = -55635876.874725
[1,0] = -19013014.723410
[1,1] = -10076571.205759
[1,2] = -16736834.394614
[1,3] = -16859454.860247
[2,0] = -37112264.944333
[2,1] = -20226224.025727
[2,2] = -29240187.835329
[2,3] = -37066900.610198
[3,0] = -106091720.624028
[3,1] = -167961187.229251
[3,2] = -102707804.684645
[3,3] = -146056286.011580

Para el algoritmo con Pthread:

Matriz C

[0,0] = -58792766.396836
[0,1] = -44083996.978909
[0,2] = -56297514.088618
[0,3] = -55635876.874725
[1,0] = -19013014.723410
[1,1] = -10076571.205759
[1,2] = -16736834.394614
[1,3] = -16859454.860247
[2,0] = -37112264.944333
[2,1] = -20226224.025727
[2,2] = -29240187.835329
[2,3] = -37066900.610198
[3,0] = -106091720.624028
[3,1] = -167961187.229251
[3,2] = -102707804.684645
[3,3] = -146056286.011580

Para el algoritmo con OpenMP:

Matriz C

[0,0] = -58792766.396836
[0,1] = -44083996.978909
[0,2] = -56297514.088618
[0,3] = -55635876.874725
[1,0] = -19013014.723410
[1,1] = -10076571.205759
[1,2] = -16736834.394614
[1,3] = -16859454.860247
[2,0] = -37112264.944333
[2,1] = -20226224.025727
[2,2] = -29240187.835329
[2,3] = -37066900.610198
[3,0] = -106091720.624028
[3,1] = -167961187.229251
[3,2] = -102707804.684645
[3,3] = -146056286.011580

Como se puede apreciar, los resultados fueron idénticos, demostrando la correctitud de los algoritmos paralelos (adjunto los archivos .c con “print” como nombre al final por si ustedes quieren hacer la prueba de los resultados)

Comparación entre modelos

Comparación de las mejores versiones de cada modelo:

N (bs = 64)	Versión secuencial	Versión Pthread			Versión OpenMP		
		T = 2	T = 4	T = 8	T = 2	T = 4	T = 8
512	0.561895	0.291988	0.147274	0.075757	0.292288	0.147484	0.075324
1024	4.352408	2.230519	1.126372	0.578590	2.233557	1.126982	0.578433
2048	34.835465	17.617868	8.898398	4.521590	17.620263	8.928093	4.526614
4096	269.095233	135.592896	68.233959	34.857703	135.918132	68.412556	34.955603

SpeedUp

N (bs = 64)	SpeedUp - Versión Pthread			SpeedUp - Versión OpenMP		
	T = 2	T = 4	T = 8	T = 2	T = 4	T = 8
512	1.924377	3.815303	7.417070	1.922401	3.809870	7.459707
1024	1.951298	3.864094	7.522439	1.948644	3.862003	7.524480
2048	1.977280	3.914801	7.704251	1.977011	3.901781	7.695700
4096	1.984582	3.943714	7.719821	1.979833	3.933418	7.698200

Eficiencia

N (bs = 64)	SpeedUp - Versión Pthread			SpeedUp - Versión OpenMP		
	T = 2	T = 4	T = 8	T = 2	T = 4	T = 8
512	0.962189	0.953826	0.927134	0.961201	0.952468	0.932463
1024	0.975649	0.966024	0.940305	0.974322	0.965501	0.940560
2048	0.988640	0.978700	0.963031	0.988506	0.975445	0.961963
4096	0.992291	0.985929	0.964978	0.989917	0.983355	0.962275

Conclusión

Como conclusión podemos comentar que el uso de hilos provee un incremento altísimo de la performance. En nuestro caso los valores de speedup y eficiencia se mantienen altos a medida que aumentamos la cantidad de procesadores utilizados para resolver el problema, lo que es óptimo para lograr mejores tiempos de ejecución. El poder mantener estos valores altos se debe principalmente a que en nuestro código se encuentran varias regiones paralelizables, pero no siempre vamos a correr con la misma suerte, incluso en este problema se puede apreciar que el cálculo del promedio es algo que se hace de manera secuencial. Sin embargo, como la mayoría del código se puede paralelizar, ya que, por ejemplo, las iteraciones de los for son independientes, la mejora es notable, teniendo un SpeedUp muy cercano al óptimo.

En cuanto a las tecnologías utilizadas, se puede apreciar como los tiempos de Pthread son ligeramente mejores que los de OpenMP, lo cual es entendible ya que estamos haciendo un uso directo de las herramientas que nos provee. En cambio, cuando se utiliza la librería OpenMP, en esta conversión o pasaje a Pthread que realiza la performance se ve ligeramente afectada. Sin embargo, el código programado con OpenMP es ampliamente más fácil de escribir y de entender, lo cuál nos provee una gran ventaja a la hora de desarrollar, quedará en cada proyecto analizar cual es la conveniencia y la prioridad que se le da a la ventaja que provee cada herramienta, pero en terminos generales OpenMP se podría considerar como la mejor alternativa.