Sistemas Paralelos - TP 2 - Programación en Memoria Compartida

Autores

- Juan Cruz Cassera Botta, 17072/7
- Lautaro Josin Saller, 18696/9

Especificaciones del sistema utilizado

Cluster (usaremos el Multicore Blade)

- Nodos: 16.
- Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro núcleos cada uno a 2.0GHz.
- Caché L1 DATA: 32K.
- Caché L1 Instruction: 32K.
- Caché L2: 6144K.
- Sistema Operativo: Debian 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5+deb10u2 (2019-08-08) x86_64.

Enunciado

Dada la siguiente expresión:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times B^T]$$

- **Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.**
- **MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.**
- **MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.**
- B^T **es la matriz transpuesta de B.**

Desarrolle 3 algoritmos que computen la expresión dada:

1. **Algoritmo secuencial optimizado.**
2. **Algoritmo paralelo empleando Pthreads.**
3. **Algoritmo paralelo empleando OpenMP.**

Mida el tiempo de ejecución de los algoritmos en el cluster remoto. Las pruebas deben considerar la variación del tamaño de problema (N = {512, 1024, 2048, 4096}) y, en el caso de los algoritmos paralelos, también la cantidad de hilos (T = {2, 4, 8}).

Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.

Aclaraciones

Sobre el tamaño de bloque

En los 3 algoritmos que desarrollamos, usamos tamaño de bloque (para la multiplicación de matrices por bloque) = **128**, ya que fue el valor que mejores resultados nos dio en las pruebas del TP 1. Además, ya no se pasará como argumento al programa, si no que será una variable global.

En las soluciones paralelas, sin embargo, en el caso en que el tamaño de las porciones asignadas a cada hilo sean menores al tamaño de bloque, el tamaño de bloque se ajusta a la porción.

Por ejemplo, con N = 512 y cantidad_hilos = 8, donde la porción es 64, que es menor al tamaño de bloque 128, el nuevo tamaño de bloque ajustado será 64 y no 128.

Se tomó esta decisión debido a que con N = 512 y cantidad_hilos = 8 se generaba un error durante la ejecución del programa debido a que se generaban accesos inválidos a memoria (*segmentation fault*). Ajustar el tamaño de bloque al de la porción evita estos desbordes.

Sobre los archivos utils.c y utils.h

Además de los códigos que resuelven los cálculos con matrices, se provee un archivo **utils.c** el cual contiene funciones para imprimir matrices y testear los resultados obtenidos, y este archivo es importado por las 3 soluciones (secuencial, Pthreads, OpenMP).

Esto no implica ninguna penalización de performance, ya que estas funciones importadas no se usan dentro del bloque de código donde se mide el tiempo de ejecución.

Se incluye este archivo a la hora de compilar los 3 programas, como se muestra más abajo en este documento en la sección de **Tiempos de ejecución**, en los comandos gcc que se usaron.

Realizamos esta división para que los archivos no queden tan grandes y sean más manejables.

Explicación de los algoritmos

Algoritmo secuencial optimizado

El algoritmo secuencial que usaremos será el optimizado que desarrollamos para el TP 1 y que fue explicado ahí, con la única diferencia que el tamaño de bloque ya no se pasa como argumento.

El código de esta solución se encuentra en el archivo **secuencial.c**.

Algoritmo paralelo empleando Pthreads

Como se mencionó en la **aclaración**, el tamaño de bloque se vuelve variable para contemplar casos límite como N = 512 y cantidad_hilos = 8.

En la implementación el trabajo se reparte entre hilos asignando a cada hilo un rango contiguo de filas de las matrices de dimensión N×N. Concretamente, el hilo i procesa las filas cuyo índice va desde

inicio = i * (N / T) hasta ***fin = (i + 1) * (N / T) - 1***

De este modo, cada hilo opera sobre un subconjunto independiente de la matriz, garantizando que no haya solapamientos al escribir en el resultado. Esta forma de trabajo elimina la necesidad de sincronización en buena parte de la implementación. El único momento donde es necesario la utilización de **semáforos** es para el cálculo del mínimo, máximo y promedio de las matrices A y B. Al mismo tiempo, es necesaria la utilización de **barreras** para evitar errores al calcular el cociente y al calcular la matriz final.

Función ***main():***

1. Recibe los argumentos N y cantidad_hilos.
2. Ajusta el tamaño de bloque si la porción de las matrices que trabajará cada hilo es menor al tamaño de bloque.
3. Aloca memoria para las matrices y las inicializa de igual forma que en la solución secuencial.
4. Inicializa **2 semáforos, 2 barreras,** y declara tantos hilos como cantidad_hilos se envió por argumento.
5. **Comienza a medir el tiempo.**

6. Crea los hilos, asignándole su ID a cada uno, y donde cada uno ejecutará la función **computo_general**.
7. Espera a que todos los hilos hayan terminado de trabajar.
8. **Termina de medir el tiempo y lo imprime**.
9. Chequea que los resultados sean correctos.
10. Libera la memoria alocada a las matrices.

Función ***computo_general():***

1. Es la función que ejecuta cada hilo.
2. Define los índices de inicio y fin para cada hilo usando el ID de cada hilo (que la función main le envía), de manera de definir qué porción de cada matriz trabajará cada hilo, para evitar pisarse.
3. Define variables locales para los cálculos de máximo, mínimo y promedio de las matrices A y B.
4. Cada hilo inicializa su parte de la matriz B transpuesta.
5. Luego se hacen los cálculos para obtener los mínimos, máximos y promedios locales y se actualizan los mínimos, máximos y sumas (acumuladoras para los promedios) globales (max_A, min_A, max_B, min_B) haciendo uso de **semáforos para tener exclusión mutua.**
6. Luego los hilos **deben esperar en una barrera** porque el cálculo del cociente global requiere ya tener calculados los máximos, mínimos y promedios globales, es decir, que se hayan procesado por completo las matrices A y B, lo cual implica que todos los hilos deben haber terminado de recorrer su porción.
7. Un solo hilo calcula el cociente global (el de id 0, pero podría ser cualquier otro), ya que es innecesario que todos lo hagan.
8. Una vez finalizado el cálculo anterior, los hilos proceden a realizar las multiplicaciones entre las matrices AxB y CxB_T sin ningún impedimento ya que no es necesario ningún tipo de sincronización (cada porción de la matriz pertenece a un hilo solamente, permitiendo una paralelización plena).
9. Finalmente, los hilos deben esperar en una segunda barrera antes de acceder al cálculo final, el cual requiere que todos los hilos hayan finalizado el cálculo de las matrices a_por_b y c_por_bt.
10. Para terminar, cada hilo finaliza utilizando **pthread_exit** con status 0 y el hilo main espera la finalización de cada uno usando **pthread_join**

El código de esta solución se encuentra en el archivo **pthreads.c**.

Algoritmo paralelo empleando OpenMP

1. El programa recibe los argumentos N y cantidad_hilos. Nuevamente el tamaño de bloque es fijo (128) salvo en el caso de N = 512 donde pasa a ser 64.
2. Se definen las variables que serán usadas por todos los hilos, se inicializan las matrices y se establece el número de hilos a ser usado con la función **omp_set_num_threads.**
3. Se utiliza la cláusula ***parallel*** para definir el código que será ejecutado en paralelo por todos los hilos. A su vez se definen las variables privadas de cada hilo.
4. A continuación se transpone la matriz **B_T** usando el constructor ***for***.
5. Luego se utilizan 2 constructores ***for*** para obtener el mínimo, máximo y promedio de las matrices A y B haciendo uso de la cláusula ***reduction*** para simplificar los cálculos, la cual hace (de forma implícita) que cada hilo tenga su copia local de los escalares y al terminar OpenMP combina todas estas copias en una sola "global".
    a. Es necesario que todos los hilos esperen antes de poder calcular el cociente global, por lo que no se agrega la cláusula ***nowait***.
6. Uno solo de los hilos, sin importar cual, realiza el cálculo de los promedios a partir de las sumas de los valores de las matrices, así como el cálculo del cociente. Esto se logra usando la cláusula ***single***.
7. Después, se procede a calcular la multiplicación entre las matrices A y B haciendo uso de la cláusula ***nowait*** ya que no es necesario que los hilos esperen a que todos hayan terminado para poder calcular la multiplicación siguiente entre C y B_T.
8. Se multiplica C x B_T. Una vez terminada la multiplicación, los hilos deben sincronizarse antes de proceder con la operación final, ya que es necesario que todos hayan finalizado las dos multiplicaciones anteriores. Esto ocurre automáticamente ya que OpenMP realiza una barrera implícita si no se agrega la cláusula ***nowait***.
9. Para la operación final se decide usar nuevamente la cláusula ***nowait*** porque ya no es necesario ningún otro tipo de sincronización. Los hilos pueden terminar su ejecución una vez terminada su parte del cálculo.
10. Como en los otros dos algoritmos, el hilo main valida los resultados obtenidos luego de terminar de medir el tiempo.

11. Se libera la memoria de las matrices.

El código de esta solución se encuentra en el archivo **openmp.c**.

Tiempos de ejecución (TAM_BLOQUE = 128 excepto en el caso naranja, donde se usa 64)

Algoritmo secuencial

- Compilado con: **gcc -O3 -Wall -o sec secuencial.c utils.c**
- Enviado a ejecutar al cluster con el script: **script-secuencial.sh**

N			
512	**1024**	**2048**	**4096**
0.418919	3.410937	26.937330	214.812314

Algoritmo paralelo empleando Pthreads

- Compilado con: **gcc -O3 -Wall -pthread -o pth pthreads.c utils.c**
- Enviado a ejecutar al cluster con el script: **script-pthreads.sh**

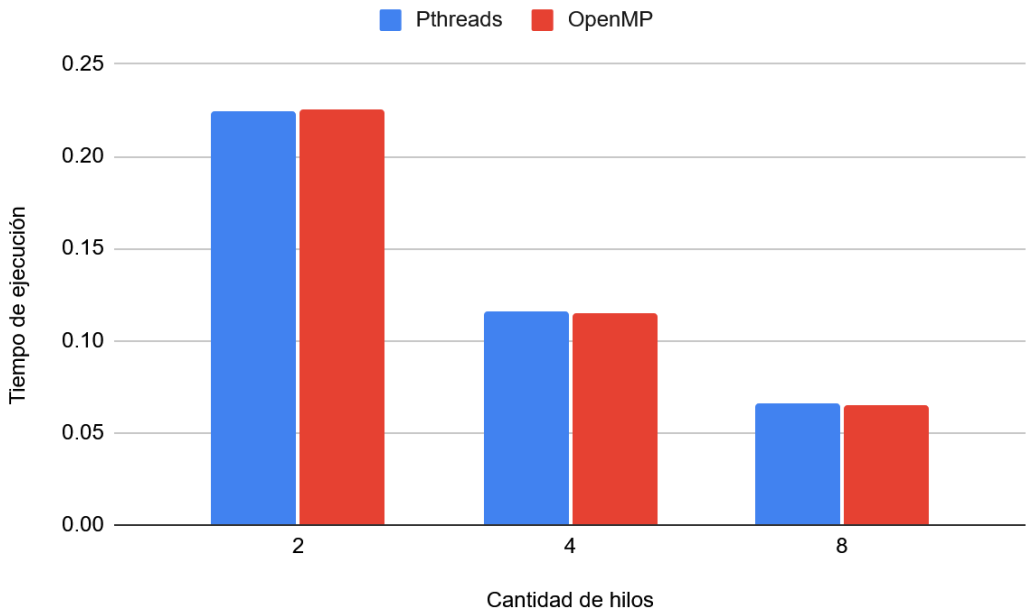
Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
2	0.224667	1.835186	14.341564	114.274786
4	0.116204	0.935988	7.270978	57.609923
8	0.065967	0.490787	4.220474	30.286527

Algoritmo paralelo empleando OpenMP

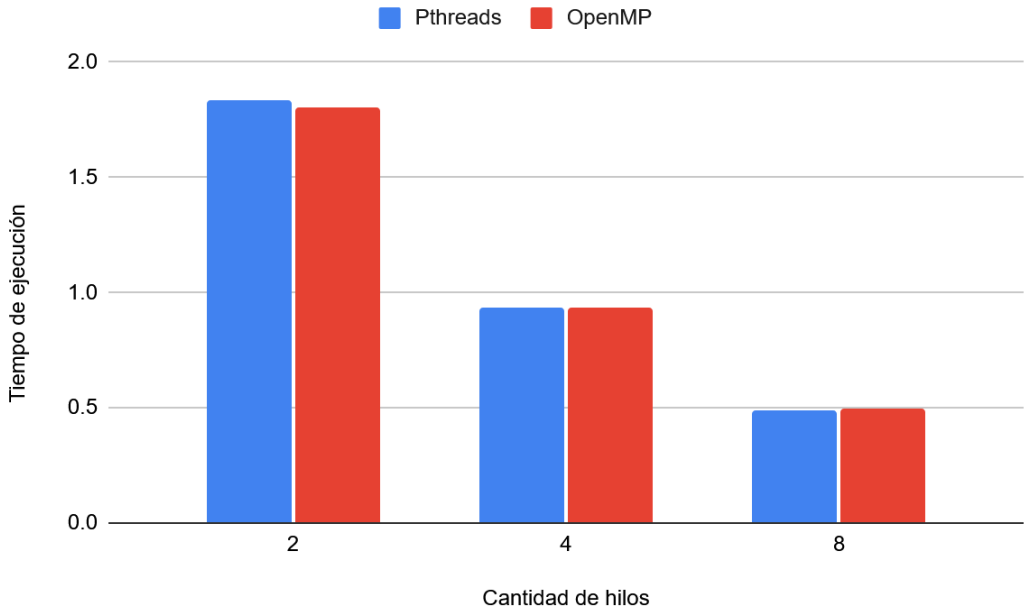
- Compilado con: **gcc -O3 -Wall -fopenmp -o open openmp.c utils.c**
- Enviado a ejecutar al cluster con el script: **script-openmp.sh**

Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
2	0.225895	1.807514	14.443807	113.856166
4	0.115249	0.934796	7.279727	57.573401
8	0.065395	0.495386	3.776709	34.445667

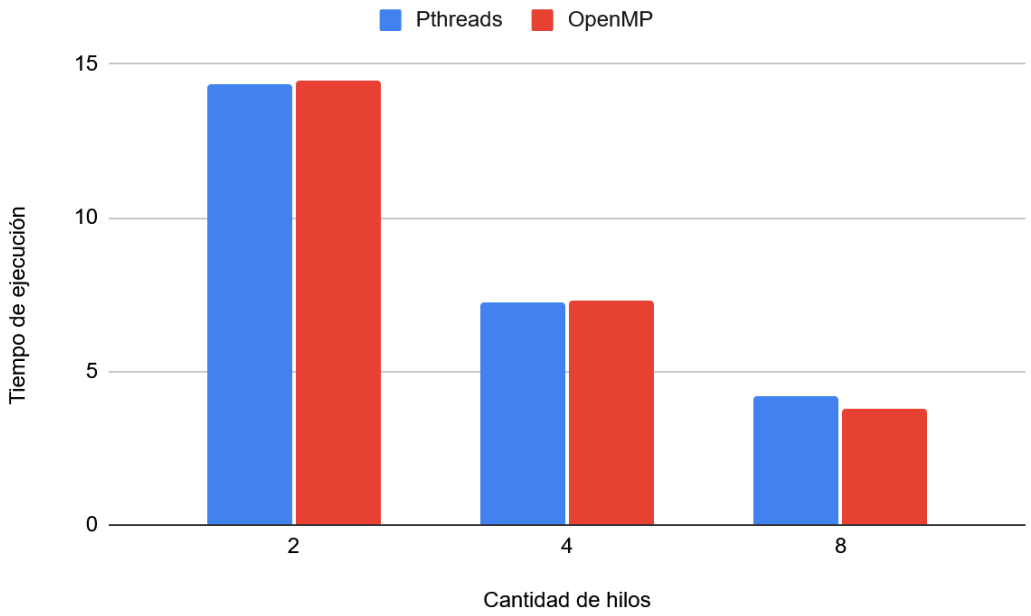
Comparación gráfica entre Pthreads y OpenMP con N = 512



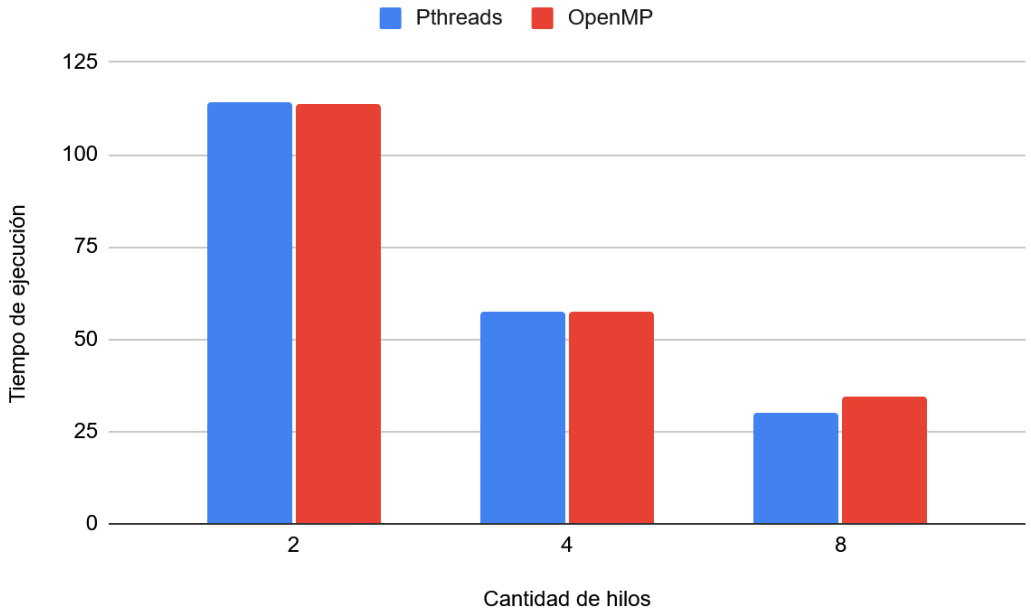
Comparación gráfica entre Pthreads y OpenMP con N = 1024



Comparación gráfica entre Pthreads y OpenMP con N = 2048



Comparación gráfica entre Pthreads y OpenMP con N = 4096



Speedup (TAM_BLOQUE = 128 excepto en el caso naranja, donde se usa 64)

El Speedup indica qué tanto beneficio obtenemos al usar la solución paralela comparado a usar la solución secuencial. Se define por la fórmula:

$$\frac{TiempoSecuencial}{TiempoParalelo}$$

donde:

- El tiempo secuencial debe obtenerse usando el algoritmo secuencial más rápido/óptimo posible para el problema dado.
- Ambos tiempos dependen del tamaño de problema (N).
- El tiempo paralelo también depende de la cantidad de hilos que se usan.
- Si el resultado del cociente es mayor a 1, la solución paralela es mejor.
- Si el resultado del cociente es igual a la cantidad de procesadores, el speedup es óptimo.
- Si el resultado del cociente es mayor a la cantidad de procesadores, el speedup es superlineal.

Algoritmo paralelo empleando Pthreads

Cantidad de hilos	N			
	512	1024	2048	4096
2	1.864621863	1.858632858	1.878270041	1.879787497
4	3.605030808	3.644210182	3.704773966	3.728738086
8	6.350432792	6.949933474	6.382536653	7.092669093

Algoritmo paralelo empleando OpenMP

Cantidad de hilos	N			
	512	1024	2048	4096
2	1.854485491	1.887087458	1.86497438	1.886698995
4	3.634903557	3.648857077	3.700321454	3.731103431
8	6.40597905	6.885412587	7.132487571	6.236265188

Eficiencia (TAM_BLOQUE = 128 excepto en el caso naranja, donde se usa 64)

La eficiencia indica qué tan bien se están aprovechando los recursos que se tienen, es decir los procesadores. En arquitecturas homogéneas, como la del Multicore Blade que usamos, se define por la fórmula:

$$\frac{Speedup}{Número\ de\ procesadores}$$

donde:

- Como nunca usaremos más hilos que la cantidad de núcleos que tiene Blade (8), si usamos X hilos, se usarán X procesadores.
- El resultado está entre 0 y 1.
- Cuanto más se acerca a 1, más se está aprovechando la arquitectura.
- Cuanto más se acerca a 0, menos se está aprovechando la arquitectura.

Algoritmo paralelo empleando Pthreads

Cantidad de hilos	N			
	512	1024	2048	4096
2	0.9323109313	0.929316429	0.9391350204	0.9398937487
4	0.901257702	0.9110525455	0.9261934914	0.9321845214
8	0.793804099	0.8687416843	0.7978170817	0.8865836367

Algoritmo paralelo empleando OpenMP

Cantidad de hilos	N			
	512	1024	2048	4096
2	0.9272427455	0.9435437291	0.9324871898	0.9433494976
4	0.9087258892	0.9122142692	0.9250803636	0.9327758577
8	0.8007473813	0.8606765734	0.8915609463	0.7795331485

Conclusión

Podemos concluir, para empezar, que las soluciones paralelas tanto utilizando Pthreads como OpenMP son **significativamente más rápidas y eficientes que la solución secuencial**, al hacer uso de múltiples unidades de procesamiento en vez de solo una. Es importante notar, sin embargo, que esta ventaja solo ocurre debido a que el cluster puede efectivamente hacer uso de 8 hilos sin overhead, ya que posee **8 cores**. Si esto no ocurriera, por ejemplo al tener **4 cores**, la mejora sería muchísimo menor, al producirse overhead por context switch en la CPU debido a tener más hilos que unidades de procesamiento.

Al comparar Pthreads con OpenMP vemos que los tiempos de ejecución son muy similares y no se observa un patrón claro, por ejemplo con N = 2048 y 8 hilos fue mejor OpenMP mientras que con N = 4096 y 8 hilos fue mejor Pthreads. En cuanto a legibilidad nos resulto más sencillo utilizar Pthreads.

Por un lado, notamos que (en Pthreads y OpenMP) al analizar el **speedup**, conforme se usan más hilos, los resultados son mejores respecto a la solución secuencial (el speedup se acerca más al número de hilos, acercándose al speedup óptimo que en este caso sería 8).

Por otro lado, al evaluar la **eficiencia**, resulta que se aprovecha más la arquitectura al utilizar 2 y 4 hilos comparado a usar 8 (debido a que el cálculo de la eficiencia se aleja del valor óptimo 1). Esto **puede** estar ocurriendo debido a la arquitectura del cluster, que se compone de dos procesadores de 4 núcleos cada uno. Quizá, al usar 8 hilos, donde 4 se ejecutan en un procesador y 4 en el otro, se está produciendo cierto nivel de overhead cuando estos hilos en dos procesadores separados necesitan comunicarse entre sí.