Entrega 2 - Sistemas Paralelos

Alumnos: Gaitan Catalina, Lizondo Santiago.

Enunciado

Dada la siguiente expresión:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times Pot2(D)]$$

Desarrolle 3 algoritmos que computen la expresión dada:

- Algoritmo secuencial optimizado
- Algoritmo paralelo empleando Pthreads
- Algoritmo paralelo empleando OpenMP

Mida el tiempo de ejecución de los algoritmos en el cluster remoto. Las prueban 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}).

Las pruebas fueron realizadas sobre un único nodo del cluster provisto por la catedra, que tiene las siguientes características:

- 8GB de memoria RAM
- Dos procesadores Intel Xeon E5405

Características del procesador Intel Xeon E5405

- CPU class: Server
- Clock Speed: 2,0GHz
- Turbo Speed: Not Supported
- Numero de cores físicos: 4
- Numero de hilos por core: 4
- Cache: L1: 256KB, L2: 12.0MB, L3: No tiene
- Intel Hyper-Threading: Not Supported
- Velocidad del bus: 1333MHz

Algoritmo secuencial

Arreglos hechos al algoritmo secuencial realizado en la entrega anterior:

- El calculo de la multiplicación de matrices se realiza con funciones para que sea mas comprensible la lactura, y mas modular. Si se optara por quitar las funciones los tiempos son menores pero la diferencia en este caso es muy minima comparado con el costo / beneficio de tener un algoritmo mas legible.

- El calculo del máximo, mínimo y promedio de las matrices A y B se realizan en diferentes fors para aprovechar la localidad de datos.
- Se utiliza un arreglo de 1..40 donde se tienen pre-cálculos las potencias de cada posición para acelerar la elevación al cuadrado de cada elemento de la matriz D.
- La multiplicación de matrices aprovecha valores pre-calculados de los indices para no tener que hacer en cada iteración la multiplicación. Por ejemplo: i*N se calcularía menos veces almacenándose en una variable auxiliar.
- En los casos en los que se debe recorrer toda la matriz y la misma se encuentra ordenada por fila, se opto por recorrerlas con un solo for hasta N*N . Esto permite ahorrar el costo de computo de multiplicación por cada posición.

Algoritmo paralelo con Pthreads

Código del main:

Lo primero que se realiza al empezar a contar el tiempo de ejecución es inicializar el arreglo con las potencias del 1 al 40, para que se ejecute de forma secuencial. Esta decision fue debido que al ser un computo tan pequeño, no es necesario ponerlo dentro de la sección paralela.

Luego se lleva a cabo la creación de los hilos usando la función "*pthread_create()*" y mandándoles como parámetros: el manejador del hilo, un atributo (que en este algoritmo no es utilizado, ya que no es de nuestro interés asignar prioridades), la función que ejecutaran todos los hilos, y el id correspondiente a cada uno.

Por ultimo se bloquea al hilo master hasta que los demás terminen (ejecuten el *pthread_exit()*), sino sucedería que el master termina su ejecución y los hilos no podrían terminar de calcular los valores.

Función calcular_ecuacion:

Es el código que ejecutara cada hilo. Dentro de ella podemos encontrar las diferentes funciones, donde a cada una se le pasa por parámetro su id. Este parámetro luego es utilizado por cada hilo para calcular el "strip" de las matrices que le correspondería a cada uno.

Aunque en algunas operaciones las distribuciones de las filas y columnas no es igual para todas las matrices. Por ejemplo en la multiplicación de matrices, la id se utiliza para calcular la primera y ultima fila que debería acceder la matriz que se esta accediendo por filas (A y C en este ejercicio), pero las matrices que se acceden por columna (B y D2) se acceden de manera completa.

Aspectos tenidos en cuenta:

- Barreras: En la solución se utilizaron la menor cantidad de barreras posibles, ya que la sincronización es uno de los factores que mas incrementan el tiempo de ejecución.

  La utilización de la barrera es obligatoria, ya que para poder calcular los promedios y el valor del escalar se va a requerir que todos los procesos hayan terminado de recorrer su porción de matriz correspondiente.

  Tambien se incluye el calculo de elevar cada elemento de la matriz D al cuadrado antes de la barrera, ya que para luego realizar la CxD, voy a necesitar D completa. Esto es porque la multiplicación de matrices depende de tener todas las columnas de la segunda matriz (D2 en este caso) para poder realizar los productos correctamente. Si se intenta multiplicar la matriz C con D2

antes de que todas las columnas de D2 estén calculadas (Como seria el caso de que no haya ninguna barrera) , algunos productos se calcularán con valores no inicializados y los resultados serán incorrectos.

- Calculo del promedio y de RP: debido a que el valor de RP, lo van a necesitar todos los hilos para realizar la multiplicación de todos los elementos de la matriz AB con RP y que son cálculos que llevan muy poco computo, se vio pertinente realizar que cada hilo calcule estos valores.
- Uso de locks: en la búsqueda de los valores máximos, mínimos, y suma de las matrices , se ha utilizado un único lock para las 3 variables.

  El uso de un lock por variable podría explotar mas el paralelismo del algoritmo, pero debido a que los valores se actualizan casi en un mismo instante de tiempo y que dentro del lock no se realiza mucho computo, el uso de 3 locks terminarían perjudicando el tiempo de ejecución del programa

> En resumen, se emplea 1 sola barrera por dos motivos:
>
> 1. Por la multiplicación de C x D2 que necesita que se termine de elevar al cuadrado todos los valores de D
> 2. Por el calculo de RP que depende que ya se hayan encontrado todos los valores de máximo , mínimo y promedio de A y B. Como estos dos puntos no dependen uno del otro se puede aprovechar 1 barrera para solucionar ambos.

Algoritmo paralelo con OpenMP

Al igual que en Pthreads, la creación y destrucción de los hilos se hace una única vez, ya que realizar estas acciones reiteradas veces agregaría un overhead adicional.

Por esta razón se realiza utiliza una única vez el constructor parallel, que va a crear la cantidad de hilos que fueron inicializados con la función “*omp_set_num_threads ();”*

Gracias a que el estándar basado en directivas OpenMP está creado específicamente para la programación paralela, el algoritmo se han simplificado considerablemente:

- Directiva for: gracias a esta directiva, la distribución de datos entre los hilos ya no es responsabilidad del programador.
- Clausula reduction: esta clausula nos permite poder encontrar el máximo, mínimo y la suma de los valores de las matrices, sin tener que encargarnos de usar locks o semáforos.
- Constructor single: este constructor nos permite fácilmente crear una sección critica, donde solo un hilo ejecutará el código que se contiene dentro. En nuestro algoritmo es útil para que solo un hilo calcule los promedios y el valor de RP.
- Directivas nowait: debido a que openMP tiene barreras implícitas en las directivas o regiones paralelas, se opto por utilizar “*nowait*” para que no sea necesario esperar a que los otros hilos completen sus iteraciones antes de continuar con la siguiente iteración. Esto permite una mayor utilización de los recursos y una ejecución más rápida del programa

  En algunas directivas no es posible poner “*nowait*” debido a que si o si se necesitan que algunas operaciones se terminen de realizar para poder empezar las demás (calculo de RP y elevar al

cuadrado cada elemento de D).

- schedule(static): Esta clausula hace que las iteraciones se asignan estáticamente a los hilos de manera equitativa. Se ha elegido utilizar esta clausula por las siguientes razones:
  1. Las matrices son cuadradas , tienen el mismo numero de filas que de columnas y se dividen en bloques de tamaño fijo para cada thread
  2. Se cuenta con nodos con 2 procesadores de 4 núcleos cada uno de igual potencia.

> Tanto en la solución con Pthreads como con OpenMP contienen un check que verifica el correcto resultado de la operación.
>
> Este check únicamente funciona si las matrices están inicializadas con valor 1, pero también han sido comprobados con valores obtenidos de manera aleatoria.

Tiempos de ejecución

Los 3 algoritmos se ejecutaran con las siguientes características:

- Optimización O3: debido a que fue la que dio resultados mas favorables en cuanto a performance y precisión matemática
- Valor 8 como tamaño de bloque: a causa de que fue el valor que mejores tiempos de ejecución dio como resultado
- Se ejecuta el algoritmo 3 veces y como resultado en la tabla se pone el promedio de los valores

Tiempos de ejecución del **algoritmo secuencial:**

N	512	1024	2048	4096
Tiempo ejecución	0,6302973	5,1153553	40,600667	321,0295176

Tiempos de ejecución del **algoritmo paralelo con Pthread**:

Threads / N	512	1024	2048	4096
2	0,319165	2,592306	20,570060	161,651996
4	0,162786	1,335340	10,55118	82,775456
8	0,087954	0,710932	5,513555	43,953785

Tiempos de ejecución del **algoritmo paralelo con openMP**:

Threads / N	512	1024	2048	4096
2	0,310462	2,509158	19,915358	157,01270
4	0,158975	1,306469	10,298602	80,7673223
8	0,0850746	0,686168	5,388172	42,056858

Calculo de SpeedUp y Eficiencia

El **SpeedUp** es una de las métricas mas populares. Refleja el beneficio de emplear procesamiento paralelo para resolver un problema dado comparado a realizar en forma secuencial.

Para calcular el SpeedUp se realiza la siguiente division:

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

Nota: Para el calculo del tiempo secuencial se utilizo el algoritmo mas optimizado, que se ha explicado anteriormente

La **eficiencia** es la medida de la fracción de tiempo en la cual las unidades de procesamiento son empleadas en forma util. Para calcularlo se realiza la siguiente division:

$$E_p(n) = \frac{S_p(n)}{S_{optimo}}$$

- Como la arquitectura del nodo del cluster donde estamos corriendo los algoritmos es homogénea, el SpeedUp Optimo va a ser la cantidad de threads que le asignemos al código paralelo

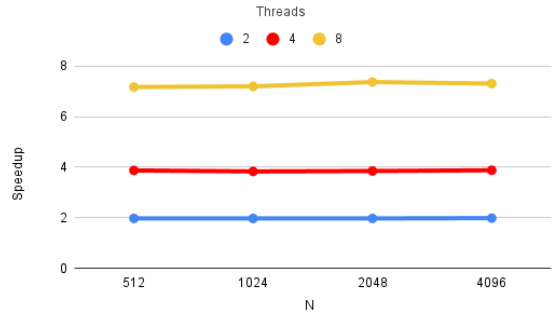
Pthreads

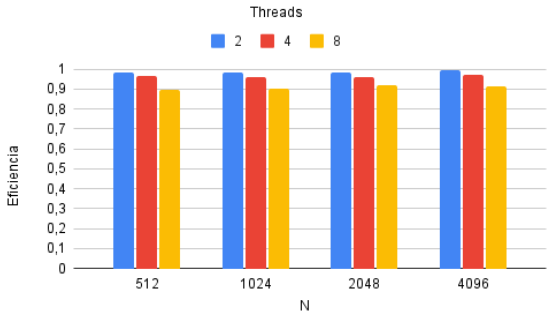
SpeedUp

	Order of matrix			
P	512	1024	2048	4096
2	1,97	1,97	1,97	1,99
4	3,87	3,83	3,85	3,88
8	7,17	7,20	7,36	7,30

Eficiencia

	Order of matrix			
P	512	1024	2048	4096
2	0,985	0,985	0,985	0,995
4	0,968	0,958	0,963	0,970
8	0,896	0,900	0,920	0,913





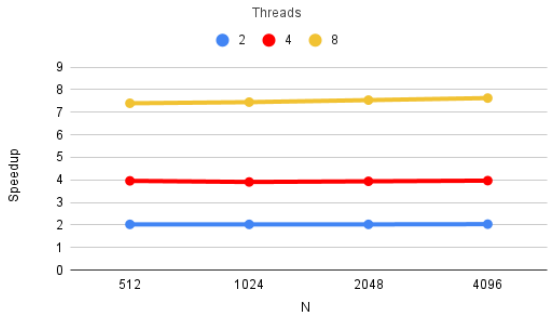
OpenMP

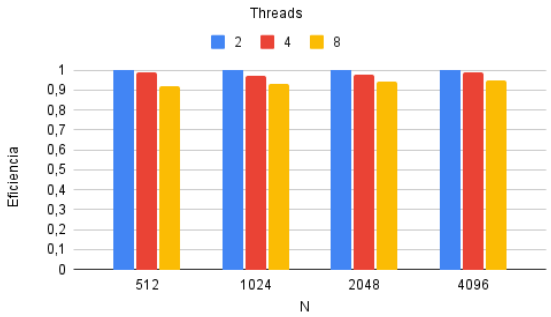
SpeedUp

	Order of matrix			
P	512	1024	2048	4096
2	2,03	2,03	2,03	2,04
4	3,96	3,91	3,94	3,97
8	7,4	7,45	7,54	7,63

Eficiencia

	Order of matrix			
P	512	1024	2048	4096
2	1,01	1,01	1,01	1,02
4	0,99	0,97	0,98	0,99
8	0,92	0,93	0,94	0,95





Análisis de los datos obtenidos

A causa de la naturaleza del problema planteado, el algoritmo es altamente paralelizable (debido a que contiene poco computo secuencial y numerosas instrucciones independientes que pueden ejecutarse simultáneamente).

Gracias a esta característica puede suceder que:

- El SpeedUp sea superlineal solo por unas pocas centésimas (como sucede en el caso de OpenMP, con la utilización de 2 hilos).
- Los valores del SpeedUp siempre empiezan con valores cercanos al del SpeedUp Optimo.

En cuanto al SpeedUp tanto de Pthreads como de OpenMP:

Al incrementar el P (procesadores), y mantener fijo el N, el **speedUp mejora**. Esto es porque al utilizar una mayor cantidad de unidades de procesamiento la solución al problema se obtiene en un tiempo menor.

Al aumentar el tamaño del problema (N) y mantener el P también **mejora el speedUp** (salvo en caso excepcionales). Esto sucede porque la proporción de computo crece mas que la proporción de comunicación y sincronización entre los hilos.

En cuanto la Eficiencia tanto de Pthreads como de OpenMP:

Al mantener fijo el P y aumentar el N, la **eficiencia mejora.** Esto sucede porque el overhead que nos proporciona la comunicación y sincronización de hilos va disminuyendo en relación a la cantidad de computo que realiza cada uno.

Cuando mantengo fijo N y aumento el P, la **eficiencia empeora**. Por mas que el speedUp mejore, como es mayor la cantidad de procesos que tienen que sincronizarse y comunicarse, la proporción de overhead crece mas con respecto a la del computo que realiza cada hilo.

Conclusión

Basándonos en los resultados obtenidos, se puede concluir que los tiempos de ejecución utilizando OpenMP fueron mejores que los de Pthreads. Se podría pensar que como OpenMP utiliza internamente POSIX Threads, los tiempos de ejecución deberían ser mas lentos. Pero esto no sucede, debido a que seguramente el uso de esta librería es mas eficiente en OpenMP, que en el algoritmo planteado en el informe. Sin embargo, las diferencias son mínimas y los tiempos son bastante similares. Es importante destacar que el proceso de desarrollo con Pthreads es más laborioso y requiere prestar atención a más detalles que con OpenMP gracias a sus directivas.

En cuanto a la eficiencia, ambos logran el mayor nivel de eficiencia cuando se utilizan solo dos hilos, y la eficiencia disminuye a medida que se agregan hilos (4 y 8). Cuando se utiliza una matriz de tamaño pequeño y 8 hilos, los tiempos de overhead son mayores y, por lo tanto, la eficiencia disminuye. Sin embargo, en general, los números de eficiencia son bastante altos. Esto podría deberse a que el problema es altamente paralelizable, donde la sobrecarga es muy baja.

Por lo tanto, se puede concluir que si se desea trabajar de la manera más eficiente posible para matrices de tamaños 512, 1024, 2048 y 4096, lo mejor es utilizar dos hilos y OpenMP. Por otro lado, si solo se busca lograr la mayor velocidad en todos los tamaños de matrices, es recomendable utilizar 8 hilos pero sacrificando la eficiencia.