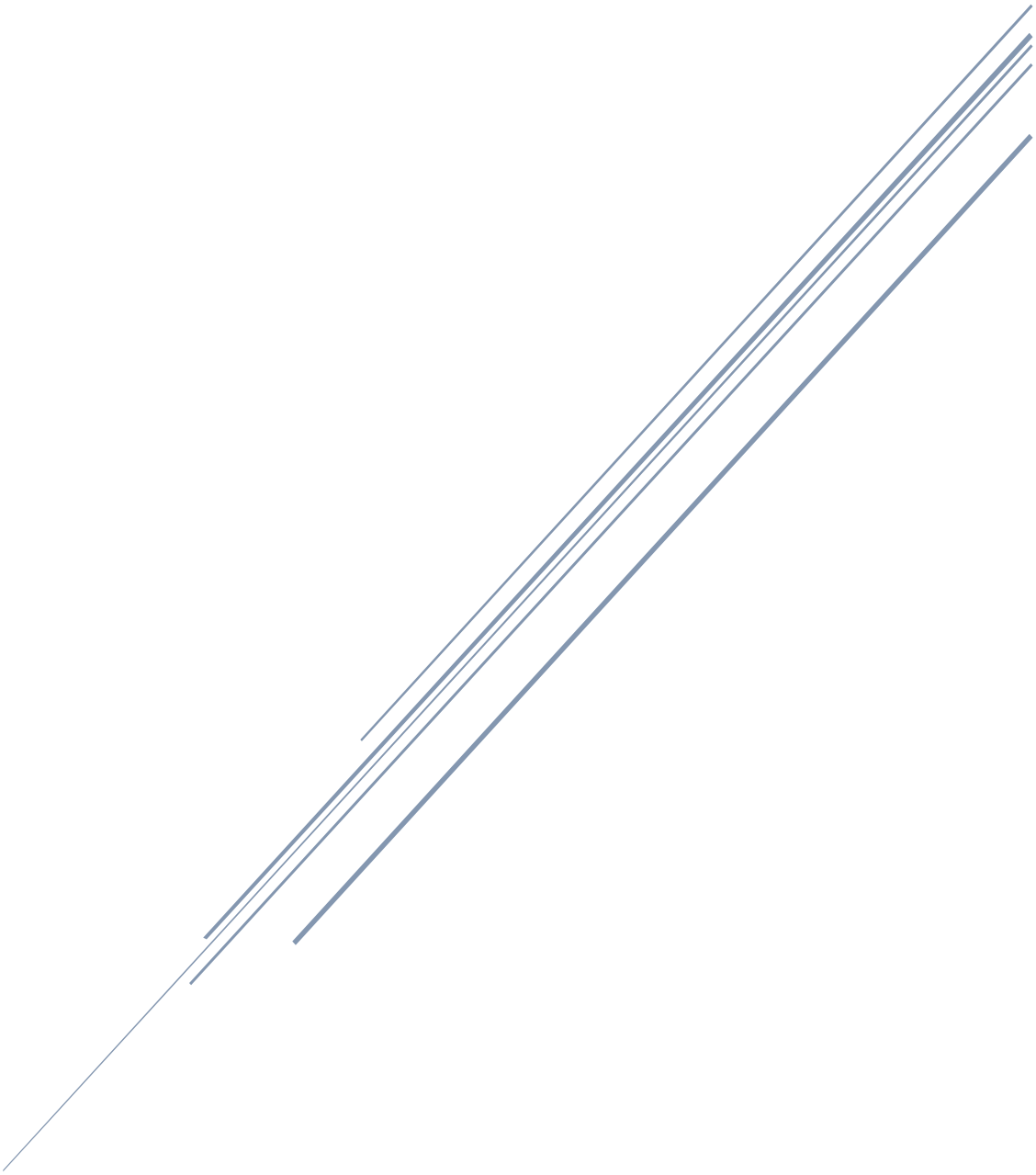
INFORME TRABAJO DE PROMOCIÓN

Sistemas Distribuidos y Paralelos 2023



Paladino Gabriel Agustin
Blanco Valentin Nicolas

Ejercicio 1

Algoritmo de multiplicación por bloques

La estrategia utilizada para realizar la multiplicación consta de 6 bucles for. Los 3 bucles mas externos se usan para iterar sobre las matrices en incrementos de tamaño de bloque bs. Los 3 bucles for más internos se encargan de realizar la multiplicación de los bloques obtenidos de cada matriz.

El objetivo de esta técnica es dividir las matrices de entrada en bloques más pequeños y realizar las operaciones de multiplicación y suma en estos bloques más pequeños en lugar de en la matriz completa. Esto puede ayudar a mejorar el rendimiento al aprovechar mejor la memoria caché y reducir los accesos a la memoria principal.

Elección del tamaño de bloque

Se eligió 128 de tamaño de bloque estándar debido a que para distintos tamaños de matrices era con el que se obtenía mejores resultados en promedio. Esto se puede ver en la siguiente tabla.

Tamaño de matriz	tamaño de bloque					
	16	32	64	128	256	512
1024	11,8407	11,1701	11,0434	10,7744	10,6984	10,6747
2048	93,375	87,9843	87,3384	86,12	85,5463	88,8273
4096	739,4443	698,5382	698,4001	688,8208	720,4238	713,4215
Promedio	281,5533	265,8975	265,5940	261,9051	272,2228	270,9745

Tabla 1: tiempos en multiplicación por bloques

Además, cabe aclarar que más adelante para la realización de la mayoría de los test se notara que se utilizaron bloques de 128 salvo algunas excepciones ya que a la hora de multiplicar en caso de usar tamaños de bloque de 128 en conjunto a algunos tamaños de matrices el primer for de filas no sería necesario ya que haría una sola iteración.

Ejercicio 2

Secuencial

Detalles sobre su implementación

Antes de programar paralelamente el algoritmo se realizó una prueba sobre qué forma era la óptima para modularizar y optimizar el código. Llegamos a la conclusión de que la mejor manera de modularizar era la siguiente:

- Procesando A*B, C*D, el máximo de D (maxD) y el mínimo de A (minA) en una primera función. Esta se realizo a partir de la función del ej1, agregando una segunda multiplicación dentro de los bucles for junto con la asignación de el mínimo y máximo correspondiente.
- Después otra función que multiplique el resultado de A*B(mattemp1) con C obteniendo Result1. También que multiplique el resultado de C*D(mattemp2) con B obteniendo Result2.
- La siguiente función realiza el cálculo de P multiplicando A*B*C(Result1 de la anterior función) por el máximo de D(maxD) y multiplicando D*C*B(Result2 de la anterior función) con el mínimo de A. Esto se hizo mediante dos bucles for donde se hacían las correspondientes multiplicaciones para luego ser sumadas en la variable P obteniendo de esta manera una vez terminado los bucles dicha matriz P
- Ya teniendo esta matriz P se saca su promedio para luego realizar la última operación. Esta función consta de sumar mediante dos bucles el total de P y almacenarlo en una variable que luego será dividida entre la cantidad de elementos
- Habiendo obtenido el promedio de P solo resta multiplicar la matriz P con dicho promedio para calcular la matriz R. Esta funcion fue realizada mediante dos bucles y multiplicando nuestra funcion P con el promedio de P de manera similar a como se multiplico ABC con maxD o DCB con minA.

Estas funciones fueron optimizadas de manera que donde se utilicen bucles for y se utilicen índices que se repiten constantemente se usen variables auxiliares para tener que evitar calcular varias veces el mismo valor y también para evitar acceder a memoria cada vez que se necesite usar.

Resultados obtenidos

En la siguiente tabla se muestran los tiempos obtenidos con el algoritmo secuencial para distintas combinaciones de N y BS. En la misma se observarán no solamente cálculos de BS para 128 sino que también para bloques de tamaño 64 y 32 para tamaño de problema de 1024 y bloques de 64 para tamaño de problema de 2048 por lo mencionado en el ejercicio 1.

Algoritmo	N	BS	Tiempo
Secuencial	1024	32	41,1297
	1024	64	40,9619
	1024	128	40,5426
	2048	64	341,83
	2048	128	338,238
	4096	128	2708,7522

Tabla 2: Tiempos ejercicio2 secuencial

Se hicieron test con esas combinaciones de N y BS ya que las mismos serán usados más adelante para calcular las métricas más adelante.

Memoria compartida

En este modelo múltiples procesadores o núcleos comparten un espacio de memoria común. De esta manera, los procesadores pueden acceder a las mismas áreas de memoria y compartir datos directamente entre ellos lo cual se mencionara proximamente.

Pthreads

Detalles sobre su implementación

Para resolver paralelamente con esta herramienta se hicieron uso de barreras para separar las dependencias de datos entre las distintas secciones de código ya que, por ejemplo, para sacar el mínimo de una matriz se necesita primeramente que todos los hilos hayan sacado el mínimo de su parte de manera local.

Una vez inicializadas las barreras se crearon la cantidad de hilos con los que se debe resolver el problema. Cada uno de ellos ejecutará el proceso resultadoR donde a partir de su id (pasado como parámetro a cada uno) operará con las partes de las matrices que le correspondan.

Dentro de la función mencionada se ejecutan las funciones definidas en el código secuencial en el mismo orden y con el mismo pasaje de parámetros. Si bien se podrían resolver las funciones sin pasaje de parámetros se optó por mantenerlo para no reducir el overhead generado y de esta forma no obtener speedups superlineales.

Dentro de las funciones secuenciales (ahora paralelizadas) se procesó de la siguiente manera:

- Para los for realizados sobre la multiplicación estos se resolvieron haciendo que cada hilo divida la matriz ordenada por filas en submatrices (esto significa paralelizando el for más externo lo que divide la matriz en 2 subbloques mínimamente por cada hilo, por esto fue utilizado tamaño de bloques de 64

para 8 hilos) mientras que la dividida por columnas fue utilizada de manera completa.

- Para los for realizados para cálculos de promedio y productos escalares los mismos se realizaron dividiendo la matriz en filas (de igual manera que el anterior esto realizado a partir de paralelizar el for más externo) haciendo que cada hilo le toque una porción de filas correspondiendo a su id de hilo junto al tamaño del problema y cantidad de hilos.
- Para sacar el mínimo de A y el máximo de D se realizó de forma que cada proceso guarde en dos arreglos compartidos sus valores calculados (uno para el máximo y otro para el mínimo). Una vez calculado esto por un proceso se compararán sus resultados con el de otro proceso (esta comparación se realiza haciendo que el proceso se compare con el de su mitad sumado a su tid). Además, para poder realizar esta comparación debemos asegurarnos de que cada proceso tenga sus resultados por lo que se usa una barrera. Para finalizar el resultado será obtenido por el hilo cero en su posición correspondiente en ambos arreglos.
- Para calcular el promedio de P se usó la misma estrategia, pero con la diferencia de que en esta se realizaron las sumas correspondientes de cada porción de matriz, de manera que la suma completa estará en el hilo cero para que luego este calculase dicho promedio.

Resultados obtenidos

Pthreads Tiempos	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
Secuencial	40,5426	338,238	2708,7522
4	10,141	84,94723	677,6336
8	5,1355	42,7526	342,3343

Tabla 3: Tiempos Pthreads

Pthreads Speedup	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	3,9979	3,9817	3,9974
8	7,9762	7,9115	7,9126

Tabla 4: Speedups Pthreads

Pthreads Eficiencia	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	0,9995	0,9954	0,9993
8	0,9970	0,9889	0,9891

Tabla 5: Eficiencias Pthreads

Análisis de escalabilidad

Como se ve en la tabla de speedup (tabla 4) de este se puede observar que a medida que aumento el número de procesadores el speedup aumenta de manera proporcional independientemente del tamaño del problema lo cual nos indica que paralelizar de acuerdo con este mecanismo dará un algoritmo fuertemente escalable. Con respecto a la eficiencia (tabla 5) se puede observar que se mantiene constante al incrementar el número de unidades de procesamiento lo cual reafirma nuevamente el hecho que sea fuertemente escalable.

OpenMP

Detalles sobre su implementación

Este código es casi idéntico al secuencial solo que se utilizan las directivas que provee openMP para paralelizar los bucles for y realizar las reducciones de variables de interés. La paralelización de los bucles for las hacemos a través de la directiva "#pragma OMP parallel for" que lo que hace es dividir el tamaño del for por la cantidad de hilos que tengamos de modo que cada hilo procese la misma cantidad de datos.

A la hora de realizar las multiplicaciones paralelizamos los fors más externos de manera tal que cada hilo realice la multiplicación de una o más filas de tamaño de bloque optimo. En las directivas para parelelizar los fors algunas variables se definieron como private ya que se necesitaba que fueran privadas para cada hilo. Además no fue necesario el uso de la cláusula shared para compartir datos entre la región secuencial y la región paralela ya que, por defecto, todas las variables se comparten excepto las declaradas.

Para calcular el mínimo de A se utilizó la cláusula reduction junto a la operación min para reducir todos los mínimos obtenidos por los hilos y así obtener el mínimo global de la matriz A.

Para obtener el máximo de D se hizo de manera análoga al mínimo de A solo que ahora se usó la cláusula reduction junto a la operación max para reducir todos los máximos calculados por los hilos y así obtener el máximo global.

Para realizar el promedio se paralelizo el for más externo de modo tal que cada hilo realice la suma de todos los elementos de una fila de altura N/T, donde N es el tamaño de la matriz y T es la cantidad de hilos que se están usando. También junto a la hora de paralelizar el for se utilizó la cláusula reduction junto a la función "+" para que se contabilice la suma total de todos los procesos en una variable. Finalmente, la suma total se divide por la cantidad de elementos de la matriz P obteniendo así su promedio.

Resultados obtenidos

openMP Tiempos	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
Secuencial	40,5426	338,238	2708,7522
4	10,2100	85,4430	681,8550
8	5,1781	42,8091	342,061

Tabla 6: Tiempos openMP

openMP Speedup	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	3,9709	3,9586	3,9726
8	7,9106	7,9011	7,9189

Tabla 7: Speedup openMP

openMP Eficiencia	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	0,9927	0,9897	0,9932
8	0,9888	0,9876	0,9899

Tabla 8: Eficiencia openMP

Análisis de escalabilidad

Como se ve en la tabla de speedup (tabla 7) y eficiencia (tabla 8) ocurre lo mismo al caso de pthreads es decir para el speedup se puede ver también un aumento

proporcional respecto al aumento de unidades de procesamiento indicando una escalabilidad fuerte. Para la eficiencia también puede decirse lo mismo que ocurre en pthreads, en la tabla de eficiencia de openmp se observa como la misma se mantiene constante al incrementar el número de unidades de procesamiento, en conclusión, este mecanismo también es fuertemente escalable.

Memoria distribuida

A diferencia del anterior modelo en este cada procesador o nodo de procesamiento tiene su propia memoria local y no comparte un espacio de memoria común por lo que cada procesador opera de forma independiente y tiene su propia visión local de los datos almacenados en su memoria.

MPI

Detalles sobre su implementación

Para paralelizar con este mecanismo primeramente se inicializo el ambiente, se obtuvo el identificador de cada proceso y se obtuvo el número de procesos para luego hacer uso de dos funciones proceso. El proceso0 que solo será ejecutado por el hilo 0, en el cual, se divide y envía la parte que tiene que procesar cada hilo, esto ocurre haciendo uso de la sentencia MPI_Scatter y MPI_Bcast, la primera seccionando la matriz ordenada por filas (enviando una porción de la misma a cada hilo) y la segunda enviando la matriz ordenada por columnas de manera completa a todos los procesos. Este proceso no solamente realiza el envió de los datos sino también procesa su parte (en donde ejecutara cada función correspondiente a su parte) y luego recibe los resultados del procesamiento de cada hilo obteniendo así la matriz P y la matriz R mediante el uso de la sentencia MPI_Gather.

Luego se encuentra el proceso1, que se ejecutara por todos los hilos restantes. En este, cada hilo recibe la parte de matriz con la que debe operar (parte de matrices ordenadas por filas y matrices ordenadas por columnas), los procesa y luego envía los resultados necesarios (matriz P y R solicitadas por el enunciado) al hilo 0 para que los junte.

A la hora de realizar las multiplicaciones la matriz ordenada por filas fue seccionada de acuerdo a la cantidad de hilos mediante un scatter como se explicó anteriormente de manera que todos hagan la misma cantidad de trabajo. Para paralelizar el for esta vez al tener un modelo de memoria distribuida este fue iniciado su recorrido en 0 sin depender ni de la cantidad de hilos tamaño del bloque ni id del hilo y el mismo fue recorrido hasta la porción que se le envio en el scatter (tamaño del problema dividido la cantidad de hilos).

Para procesar los máximo y mínimos se hizo uso de MPI_Allreduce ya que era importante que cada hilo conociera el máximo de D y el mínimo de A para calcular su porción de P que más adelante enviaría a través de un gather al proceso0 (root).

En el cálculo del promedio se usó la misma sentencia (Allreduce) que para procesar mínimos y máximos esto es de la forma para que todos conozcan el promedio haciendo que se cada código procese el promedio y ahorrándonos una comunicación ya que de la otra manera se debería

utilizar un reduce para que el proceso0 (root) haga el cálculo del promedio y luego este tenga que enviar al resto el promedio realizado.

Resultados obtenidos

MPI Tiempos	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
Secuencial	40,5426	338,238	2708,7522
4	10,4838	90,0808	721,7499
8	5,4795	43,1695	346,3487
16	2,9901	22,4582	197,4164

Tabla 9: Tiempos MPI

MPI Speedup	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	3,8672	3,7548	3,7530
8	7,4755	7,8351	7,8209
16	13,7553	15,2207	13,7210

Tabla 10: Speedups MPI

MPI Eficiencia	Tamaño de problema (N)		
Unidades de procesamiento	1024	2048	4096
4	0,9668	0,9387	0,9383
8	0,9344	0,9794	0,9776
16	0,8597	0,9513	0,8576

Tabla 11: Eficiencias MPI

Análisis de escalabilidad

Como se observa de la tabla de speedup (Tabla 10) se ve que este comienza a decaer a medida que aumenta el número de procesadores esto significa que no se ve una escalabilidad fuerte. Además, la tabla de eficiencias (Tabla 11) nos indica también que esta no se mantiene constante a medida que se incrementa el tamaño de problema junto con el número de unidades de procesamiento debido a que este decae de 0.9794 con 8 unidades y 2048 de tamaño de problema a 0.8576 con 16 unidades y 4096 de tamaño de problema. Por esto y lo anterior puede concluirse que dicho mecanismo aplicado a este algoritmo no es escalable.

Conclusión

Por todo lo anteriormente hablado para nuestro caso paralelizar este algoritmo fue más sencillo a partir del mecanismo OpenMP debido a su sintaxis más sencilla y más de "alto" nivel, lo que facilitó la implementación del código en paralelo. Por su contraparte Pthreads fue la que nos resultó más difícil de paralelizar ya que en la misma se debían implementar sentencias como los reductions del openmp.

Por otro lado, de acuerdo al análisis de escalabilidad los resultados indicaron que la estrategia de paralelización basada en OpenMP y Pthreads es la más adecuada para abordar este problema, ya que ofrece una escalabilidad fuerte y permite un mejor rendimiento a medida que se agregan más hilos de ejecución mientras que MPI no demostró ser una opción escalable en este contexto.

En resumen, de todo esto podemos decir que la estrategia que nos pareció más optima fue la de openMP debido a su practicidad y escalabilidad seguido de la estrategia de Pthreads que mostro también una fuerte escalabilidad, mientras que la implementación de MPI resultó no escalable, y de estos análisis se pueden destacar la importancia de seleccionar el enfoque adecuado de paralelización según las características del problema y los recursos disponibles.