**𝐷 = (𝑚𝑎𝑥𝐴 ⋅ 𝑚𝑎𝑥𝐵 ⋅ 𝑚𝑎𝑥𝐶 − 𝑚𝑖𝑛𝐴 ⋅ 𝑚𝑖𝑛𝐵 ⋅ 𝑚𝑖𝑛𝐶**

**---------------------------------------------------------**

**𝑎𝑣𝑔𝐴 ⋅ 𝑎𝑣𝑔𝐵 ⋅ 𝑎𝑣𝑔𝐶 ) ⋅ 𝐴𝐵𝐶**

**donde:**

**- A, B , C y D son matrices de NxN.**

**- minA y maxA son el mínimo y el máximo valor de los elementos de la matriz A,**

**respectivamente; ídem para las matrices By C.**

**- AvgA es el valor promedio de los elementos de la matriz A; ídem para B y C.**

**calcular el tiempo para secuencial**

**para paralelo:**

**con pthreads y con openmp con 2,4 y 8 hilos**

**para matrices de 512\*512 1024\*1024 2048\*2048 y 4096\*4096**

Para inicializar las matrices A, B y C hicimos:

|  |
| --- |
| for(int i=0;i<N;i++){  for(int j=0;j<N;j++){  A[i\*N+j]=i+j;  B[j\*N+i]=i+j;  C[j\*N+i]=i+j;  }  } |

Para probar que las multiplicaciones de las matrices se hicieran bien usamos los siguientes valores que se muestran abajo para inicializar las matrices (después lo sacamos para que quede mas prolijo el programa).

|  |
| --- |
| for(int i=0;i<N;i++){  for(int j=0;j<N;j++){ if((i == 0) && (j == 0)){  A[i\*N+j] =1 ;  B[j\*N+i] =2;  C[j\*N+i] =3;  }else if ((i== N-1) && (j == N-1)){  A[i\*N+j] = 10;  B[j\*N+i] = 11;  C[j\*N+i] = 12;  }else{  A[i\*N+j] =5;  B[j\*N+i] =6;  C[j\*N+i] =7;  }  } } |

Con esta inicialización me quedarían los siguientes valores:

|  |  |  |  |
| --- | --- | --- | --- |
| **Matriz** | **Min** | **Max** | **avg** |
| **A** | 1 | 10 | 5 |
| **B** | 2 | 11 | 6 |
| **C** | 3 | 12 | 7 |

|  |
| --- |
| for(i=0;i<N;i++){  disp1 = i\*N;  for(j=0;j<N;j++){  disp2 = disp1 + j;   if(maxB < B[disp2])maxB=B[disp2];  if(maxA < A[disp2])maxA=A[disp2];  if(minA > A[disp2])minA=A[disp2];  if(minB > B[disp2])minB=B[disp2];  avgA+=A[disp2];  avgB+=B[disp2];  aux = 0;  disp3 = j\*N;  for(k=0;k<N;k++){  aux+=A[disp1+k]\*B[k+disp3];  }  AB[disp2] = aux;  }  } |

|  |
| --- |
| for(i=0;i<N;i++){  disp1 = i\*N;  for(j=0;j<N;j++){  disp2 = disp1 + j;   if(maxC < C[disp2])maxC=C[disp2];  if(minC > C[disp2])minC=C[disp2];  avgC+=C[disp2];  aux = 0;  disp3 = j\*N;  for(k=0;k<N;k++){  aux+= AB[disp1+k]\*C[k+disp3];  }  D[disp2]= aux;  }  } |

|  |
| --- |
| for(i=0;i<N;i++){  disp1 = i\*N;   for(j=0;j<N;j++){  D[disp1+j]= D[disp1+j] \* d;  }  } |

La idea principal es dividir la tarea en 3 for, en donde el primer for se encargará no solo de hacer la multiplicación de la matriz A\*B sino también de calcular los mínimos, máximos y avg, de estas matrices

el segundo for realiza la multiplicación de AB \*C donde también se calcula mínimo máximo y avg de la matriz c.

por último el 3 for se encarga de realizar la multiplicación de la ecuación , previamente resuelta, con el resultado de la multiplicación de las matrices AB\*C.

Las optimizaciones que se tuvieron en cuenta al realizar el código,además de la comentada fueron:

Tener una variable auxiliar para la suma de cada elemento de la matriz, y usar el += al hacer la suma, por una cuestión de ahorrar operaciones, y por la manera que trabaja el compilador al usar += mejora que poner 2 veces la variable.

Además guardar las posiciones de la matriz ya sea en fila o columna para poder tener un acceso secuencial en la multiplicación.

y tambien tener variables disp, donde se guardaran la multiplicación de los indices, ya sea i\*N , j\*N e i\*N+j

**Tiempos de código secuencial:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **512** | **1024** | **2048** | **4096** |
| **Secuencial** | 1.942647 | 15.838669 | 126.547392 | 1018.630935 |

---------------------------------------------------------------------------------------------------------------------------

**opciones paralelas:**

**Con Pthreads:**

Para resolverlo con pthreads lo que hicimos fue:

* Una función llamada **multiplicación**  la cual van a ejecutar los hilos en paralelo.
* Declaramos las variables globales **MAXA, MAXB, MAXC, MINA, MINB, MIC**,**avgA,avgB,avgC** Para guardar los máximos, mínimos y el promedio de los valores de cada matriz
* Declaramos una barrera y 3 semáforos (uno para cada matriz) para sincronizar los hilos en el momento de calcular máximos, mínimos y suma globales.
* Cada hilo tiene sus propias variables locales para calcular el máximo, mínimo y promedio de valores de la parte de las matrices que le tocó calcular. Además de las variables **ID** y **desde y hasta** que se usaban para iterar sobre su parte de las matrices
* Usamos 3 for: En el **primero** los hilos calculan el máximo, mínimo y la suma de los valores de las matrices A y B, y también hacen la multiplicación de las matrices A y B guardandolo en una matriz llamada AB. Después hay un **segundo** for donde calculamos máximo, mínimo y la suma de valores de la matriz C, y además hacemos la multiplicación de las matrices AB y C. Luego del segundo **for** los hilos se frenan en una barrera para esperar a que todos terminen y luego de la barrera uno de los hilos (elegimos el hilo 0 por ejemplo) calcula la ecuación que se pedía en el enunciado. Pusimos una segunda barrera antes de que empiece el tercer **for** con el fin de que se espere a que el hilo 0 calcule la ecuación. Luego de la segunda barrera empieza el **tercer** **for** donde se hace la multiplicación D \* ecuación y se guarda en la matriz D.
* **Inicialización:** Inicializamos todas las variables min y max (sean globales o locales) con 999 y -1 respectivamente. Las matrices las inicializamos con valores que nos permitieran calcular su resultado para comprobar que el programa funcione correctamente.

**Tiempos medidos usando Pthreads:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cantThreads** | **512** | **1024** | **2048** | **4096** |
| **2** | 0.971887 | 7.924016 | 63.186400 | 504.350159 |
| **4** | 0.487608 | 3.986480 | 31.829386 | 254.930045 |
| **8** | 0.244190 | 1.997507 | 16.562951 | 127.673200 |

**Pseudocódigo de Pthreads:**

//declaramos las variables globales (minA,minB,maxA,maxB,etc)

//declaramos la funcion para calcular el tiempo

**void \*multiplicación(void \*id){**

//inicializamos variables locales (max,min, y avg para cada matriz)

//calculamos ID de hilo y los valores del vector que tiene que recorrer (desde, hasta)

//primer for de i:

//for de j

//En cada iteración actualizamos min, max y suma de A y B

//Realizamos la multiplicación de A \* B y la guardamos en AB

//fin de for de j

//fin de for de i

//segundo for de i

//for de j

//En cada iteración actualizamos min, max y suma de C

//Se realiza la multiplicación de AB y C y la guardamos en D

//fin de for de j

//fin del segundo for

//Se actualiza min, max y avg globales con el uso de 3 semáforos

//se usa una barrera para esperar a los demás hilos

//El hilo con ID cero calcula la ecuación pedida

// Segunda barrera para que los hilos esperen a que se calcule la ecuación

//tercer for de i

//for de j

//se multiplica D \* ecuación y se guarda en la matriz D

// fin de ambos for

// exit de los hilos

**}// fin de la función multiplicación**

**main(){** //inicio de main

//declaración de matrices y otras variables

//alocamos memoria para las matrices

//inicializamos las matrices

//aca se declaran y se inicializan las variables necesarias para manejar hilos (Se inicializan los semáforos, se inicializa la barrera, se crea el arreglo de hilos, etc).

//se empieza el conteo del tiempo

//for de t a cantThreads

//Se crea un hilo y se le pasa como parámetro la función multiplicación

//fin del for

//for de t a cantThreads

//se usa el join para que los hilos esperen a que los demás terminen

//fin del for

// se calcula el tiempo del algoritmo

//se destruyen los semáforos y la barrera

//se libera la memoria

**}** //fin de main

---------------------------------------------------------------------------------------------------------------------------

**con omp:**

Las formas de hacerlo con **omp** son muy parecidas a la de **pthreads**.

Utilizamos las directivas:

**#pragma omp parallel** para encapsular todo el código paralelo, una única vez así no se pierde tiempo en la creación de hilos,se declararon las variables (A,B,C,AB,D,mínimos,máximos,promedios) como shared y como privadas (i,j,k y un vector de 3 posiciones para la matriz a,otra para la matriz b, y otro para la c) donde la posición 1 era un maximo local del thread, la 2 un mínimo, y la 3 la suma para hacer el avg luego.

cuando se termina el primer **#pragma for** (es decir cuando el thread termino de hacer la parte de la multiplicación de a\*b),y el segundo **#pragma for**(se calculo el resultado de AB\*B sin guardar el total aún) es necesario que guarden sus valores locales en los globales, para esto es necesario hacer secciones críticas.

con la directiva **#omp critical** asegura que se va a ejecutar un solo thread de por vez.

se van a hacer 3 secciones críticas , cada una de estas secciones va a tener de ámbito una matriz(1 er seccion variables globales de matriz a, 2nda sección variables globales de la matriz b,etc)

luego de esto se debe hacer el cálculo de la ecuación, pero antes de esto se debe asegurar que se hayan actualizado todos los valores globales, para esto se pone una barrera con la directiva **#pragma omp barrier**

una vez que se pasa la barrera ya se puede realizar la ecuación , pero debemos utilizar otra directiva para que lo haga un único thread esta es **#omp single.**

la directiva omp single tiene por defecto la sincronización, así que no necesita que se espere en una nueva barrera.

por último queda realizar la multiplicación del resultado de la ecuación con lo que había dado la multiplicación de las matrices AB \* C y guardarlo en d en otro **#pragma for**

luego de esto se termina la sección de parallel, lo cual sería la forma de sincronizar que todos los hilos hayan terminado para poder tomar el tiempo

**Tiempos medidos usando OpenMP:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cant threads** | **512** | **1024** | **2048** | **4096** |
| **2** | 0.974965 | 7.946900 | 63.282831 | 510.017133 |
| **4** | 0.487895 | 4.003887 | 31.903390 | 256.411500 |
| **8** | 0.243282 | 2.000020 | 15.999239 | 129.149701 |

**SPEED UP Y EFICIENCIA**

**OMP:**

**speed up:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cant threads** | **512** | **1024** | **2048** | **4096** |
| **2** | 1.942647/ 0.974965=**1,9925** | 15.838669/ 7.946900=**1,993062578** | 126.547392/63.282831=**1,999711296** | 1018.630935//510.017133= |
| **4** | 1.942647/ 0.487895 =**3,981690733** | 15.838669 / 4.003887 = **3,955823179** | 126.547392/31.903390=**3,966581357** | 1018.630935/256.411500= |
| **8** | 1.942647/ 0.243282 =**7,985165364** | 15.838669 / 2.000020 = **7,919255307** | 126.547392/15.999239=**7,9095882** | 1018.630935/129.149701= |

**eficiencia:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cant threads** | **512** | **1024** | **2048** | **4096** |
| **2** | 1,9925/2=**0.99625** | 1,993062578  /2=**0.996531289** | 1,999711296/2=**0.999855648** | /2= |
| **4** | 3,981690733/4=**0.9954226** | 3,955823179 /4=**0.98895579475** | 3,966581357/4= **0.99164533925** | /4= |
| **8** | 7,985165364/8=**0.9981456** | 7,919255307/8=**0.98990691337** | 7,9095882/8=**0.988698525** | /8= |

Los tiempos del **speed up** nos dan proporcionalmente a la cantidad de threads que usemos para hacer el cálculo.

Por esta razon tambien tendremos una **eficiencia** casi perfecta muy cercana a 1.

**pthreads:**

**speed up:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cantThreads** | **512** | **1024** | **2048** | **4096** |
| **2** | 1.942647 / 0.971887 = **1.998840** | 15.838669/ 7.924016 = **1.998818** | 126.547392 / 63.349826 = **1.997596** | 1018.630935 / 504.350159 = **2** |
| **4** | 1.942647/ 0.487608= **3.984034** | 15.838669 /3.986480 = **3.973096** | 126.724206/ 31.829386 = **3.981358** | 1018.630935/254.930045 = **3.995727** |
| **8** | 1.942647 / 0.244190 = **7.955473** | 15.838669/1.997507 = **7.929218** | 126.724206/ 16.562951 = **7.651064** | 1018.630935**/** 127.673200 = **7.978424** |

**eficiencia:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **cantThreads** | **512** | **1024** | **2048** | **4096** |
| **2** | 1.998840/2= **0.99942** | 1.998818/2**=0.999409** | 1.997596/2=**0.998798** | 2/2= **1** |
| **4** | 3.984034/4=**0.996008** | 3.973096/4=**0.993274** | 3.981358/4**=0.995339** | 3.995727/4= **0.99893175** |
| **8** | 7.955473/8=**0.994434** | 7.929218/8=**0.991152** | 7.651064/8=**0.956383** | 7.978424/8=**0.997303** |

por lo cual con 2 hilos el **speedup** da en promedio aproximadamente 2, con 4 hilos da 4 y con 8 hilos 8.Por esta razón tenemos un algoritmo óptimo donde su **eficiencia** va a ser acorde a la cantidad de threads, aproximadamente , es decir ,E=1.