Universidad de Concepción

Ingenieria Civil Informatica

Tarea 2: Cómputo de distancia euclidiana punto a punto.

Autor:
Pablo Venegas

Docente:
Cecilia Hernández

Contents

1 Descripción de la solución.

El problema consiste en comparar la implementación paralela y secuencial del calculo de distancia euclidiana máxima y mínima de un conjunto de curvas representados por un conjunto de puntos discretos de distintas dimensiones. Se cuenta con distintos parametros que se cambiaran para poder apreciar que tanto varian estas implementaciones entre sus versiones secuencial y ocupando paralelismo. Luego de esto se realiza el analisis experimental para determinar aceleracion, eficiencia y costo a partir de los resultados obtenidos. Cabe destacar que el calculo de la distancioa se debe realizar punto a punto.

- M: número de grupos de curvas.
- K: número de curvas por grupo.
- N: número de puntos que componen cada curva.
- D: número de dimensiones del dominio de las curvas.

2 Explicación del codigo

Principalmente el codigo consta de tres funciones auxiliares y una funcion principal, las tres funciones se encargan de calcular la distancia euclidiana de las curvas, y las otras dos se encargan de obtener la minima y maxima distancia euclidiana.

Esta es la función que realiza el calculo de la distancia euclidiana entre dos puntos de dos rectas dentro del conjunto de rectas.

```
1  vector<float> distanciaEuclidiana(vector< vector<float>
       > &a_curve, vector< vector<float> > &b_curve,
       uint64_t pnt_count, uint64_t dim){
2          vector<float> delta(dim), a_comp(dim), b_comp(
               dim), dist(pnt_count);
3
4
5          float tmp = 0;
6  //      cout <<" abtes del paralel for " <<
       omp_get_thread_num() <<endl;
7          #pragma omp parallel for
8          for (int i = 0; i < pnt_count; ++i)
9          {
10
11 //             #pragma omp critical
12 //             cout <<" dentro del for " <<
       omp_get_thread_num() <<endl;
13
14                 a_comp = a_curve[i];
15                 b_comp = b_curve[i];
16
17                 float acc = 0;
18
19                 #pragma omp simd reduction(+:acc)
20                 for (int j = 0; j < dim; ++j)
21                 {
22                         acc = b_comp[j] - a_comp[j];
23                         delta[j] = acc;
```

```
24                  }
25
26                  #pragma omp simd reduction(+:tmp)
27                  for (int k = 0; k < dim; ++k)
28                  {
29                          tmp += delta[k] * delta[k];
30                  }
31
32                  //#pragma omp critical
33                  dist[i] = tmp;
34          }
35          return dist;
36  }
```

Estas son las funciones que se encargan de obtener la minima y maxima distancia euclidiana, en ambas se ocupa una reducción utilizando vectorización

```
37  float getMinDist(vector<float> &a, uint64_t size){
38          float minval = MIN;
39
40          #pragma omp parallel for reduction(min:minval)
41          for (int i = 0; i < size; ++i)
42          {
43                  if (a[i] < minval)
44                  {
45                          minval = a[i];
46                  }
47          }
48          return minval;
49  }
50
51  float getMaxDist(vector<float> &a, uint64_t size){
52          float maxval = MAX;
53
54          #pragma omp parallel for reduction(max:maxval)
55          for (int i = 0; i < size; ++i)
56          {
57                  if (a[i] > maxval)
58                  {
59                          maxval = a[i];
```

```
60                  }
61          }
62          return maxval;
63  }
```

La funcion principal del programa se encarga de implementar una funcion especial para inicializar las curvas con datos de manera aleatoria siguiendo las variables entregadas como argumento y asignandoles la memoria como corresponde.

```
64  int main(int argc, char const *argv[])
65  {
66          int M, K, N, D;
67
68          M = atoi(argv[1]);
69          K = atoi(argv[2]);
70          N = atoi(argv[3]);
71          D = atoi(argv[4]);
72
73  //      cout << " grupos: " << M << " kurvas: " << K <<"
        numero de puntos: " << N << "   Dimension: " << D <<
        endl;
74
75          vector< vector<float> > curvasK(N*D);
76          vector< vector< vector<float> > > gruposM(K*N*D)
               ;
77
78          TIMERSTART(TEST)
79          for (int i = 0; i < M; i++)
80          {
81                  for (int j = 0; j < K; j++)
82                  {
83                          initCurve(curvasK, D, N, (i+1)*j
                               );
84                          gruposM[j] = curvasK;
85                  }
```

Aqui finalmente es donde se lleva a cabo el calculo de las distancias y se obtienen el maximo y minimo segun corresponda entre cada par de curvas

en un grupo siendo almacenados en un vector.

```
86  vector<float> euclidiano, min_group, max_group;
87  float min_dist, max_dist;
88  vector< vector<float> > vectorDistancias;
89
90  for (int k = 0; k < K; k++)
91  {
92          for (int l = 0; l < K; l++)
93          {
94                  if (l != k)
95                  {
96                          euclidiano = distanciaEuclidiana
                               (gruposM[k], gruposM[l], N, D
                               );
97                          vectorDistancias.push_back(
                               euclidiano);
98                  }
99          }
100 }
101
102 for (int m = 0; m < vectorDistancias.size(); m++)
103         {
104                 min_dist = getMinDist(vectorDistancias[m
                        ], vectorDistancias[m].size());
105                 max_dist = getMaxDist(vectorDistancias[m
                        ], vectorDistancias[m].size());
106                 min_group.push_back(min_dist);
107                 max_group.push_back(max_dist);
108         }
109 //              cout << "min: " << sqrt(getMinDist(
        min_group, min_group.size())) << " max: " << sqrt(
        getMaxDist(max_group, max_group.size())) << endl;
110 }
111         TIMERSTOP(TEST)
112
113
114         return 0;
115 }
```

3 Análisis de resultados

En esta seccion se presentan los resultados obtenidos luego de realizar benchmarks sobre el codigo previamente expuesto, realizando cada prueba 3 veces con 1, 4, 8 y 12 threads, obteniendo como resultado el promedio de las 3 ejecuciones en cada caso.

Equipo de pruebas

- CPU AMD Ryzen 5 2600 6C/12T 3.4Ghz.
- 8Gb Ram DDR4 3200Mhz.
- SO Ubuntu 18.04 x64.
- GPU AMD RX480 4Gb MSI Gaming X.

Variacion en el numero de puntos por curva N

Las pruebas para este parametro consiste en cambiar el numero de puntos por curva entre 4 y 256 (potencias de 2) manteniendo los demas parametros fijos. M = 1, K = 100 y D = 2.

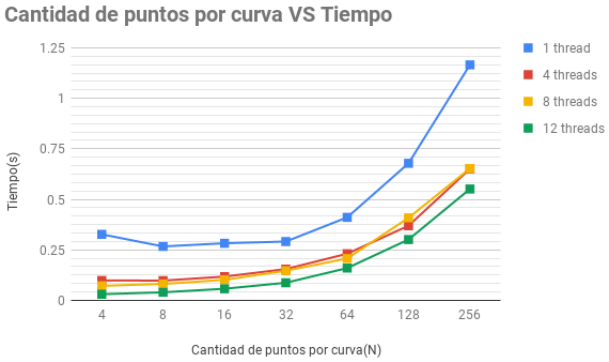

Figura 1: tiempo de ejecución vs cantidad de puntos por curva.

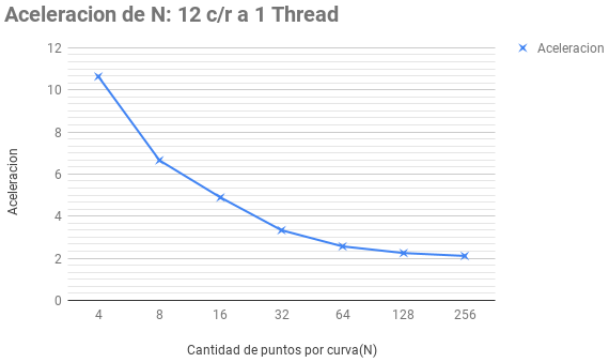

Figura 2: Aceleracion de N, 12 Threads vs 1 Thread

Variacion en el numero de curvas K

Las pruebas para este parametro consiste en cambiar el numero de curvas entre 4 y 256 (potencias de 2) manteniendo los demas parametros fijos. M = 1, N = 100 y D = 2.

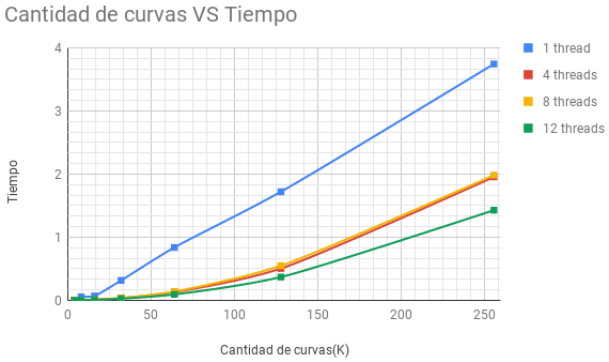

Figura 3: tiempo de ejecución vs cantidad de curvas.

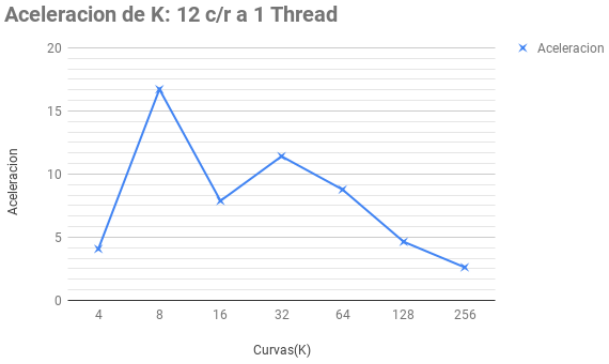

Figura 4: Aceleracion de K, 12 Threads vs 1 Thread

Variacion en el numero de grupos M

Las pruebas para este parametro consiste en cambiar el numero de grupos entre 1 y 10 manteniendo los demás parametros fijos. K = 100, N = 100 y D = 50.

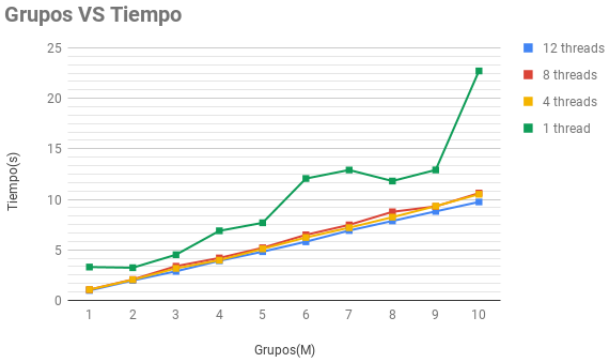

Figura 5: tiempo de ejecución vs cantidad de grupos.

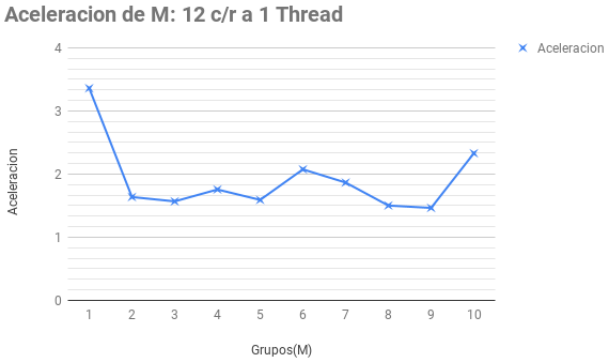

Figura 6: Aceleracion de M, 12 Threads vs 1 Thread

Variacion en el numero de dimensiones D

Las pruebas para este parametro consiste en cambiar el numero de dimensiones entre 2 y 50 (de 2 en 2) manteniendo los demás parametros fijos. M = 1, K = 100 y N = 100.

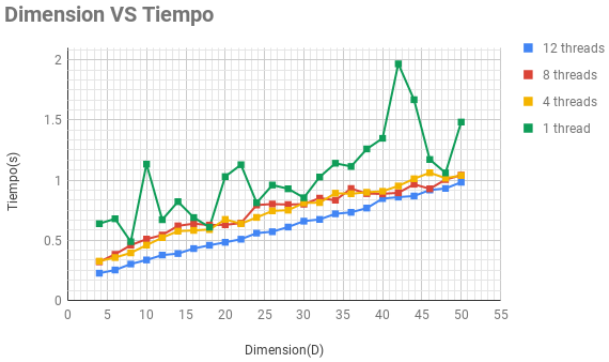

Figura 7: tiempo de ejecución vs cantidad de dimensiones.

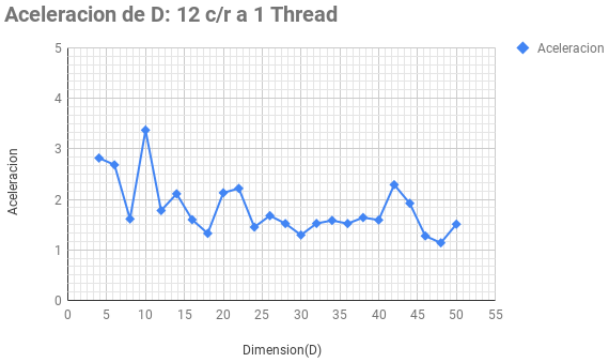

Figura 8: Aceleracion de D, 12 Threads vs 1 Thread

Discusión de resultados

A simple vista y a pesar que la cantidad de threads en que varia una implementacion y otra no se puede apreciar un gran cambio o mejoria a la hora de realizar las pruebas, solo se aprecia un gran cambio a la hora de aumentar la cantidad de puntos por curva a 256, alcanzando aqui una de las mejores aceleraciones entre los 4 casos antes planteados.

Todos estos resultados se pueden ven afectados por el planificador del sistema, dado que es posible que este obligue a las hebras a realizar muchos cambios de contexto durante la ejecución del programa, afectando de esta manera el tiempo entregado por la CPU para realizar las tareas que termina afectando tambien el rendimiento del programa.