Computació Paral·lela i Massiva

Pràctica 1.1

MIREIA GASCO AGORRETA I NEUS OLLER MATAS

Enginyeria Informàtica
Universitat Rovira i Virgili

Índex

1 Objectius

L'algoritme del veí més proper, en anglès *Travelling Salesman Problem* (TSP), és una solució heurística per al problema del viatjant de comerç. Aquest problema implica trobar la ruta més curta que passi per totes les ciutats una sola vegada i torni al punt de partida. L'algoritme comença en una ciutat arbitrària i, en cada pas, selecciona la ciutat més propera que encara no ha estat visitada com la següent en la ruta. Aquest procés es repeteix fins que s'han visitat totes les ciutats i finalment es torna a la ciutat d'origen.

Tenim com a objectiu implementar el codi seqüencial donat en un codi paral·lel utilitzant OpenMP. Això ens permetrà aconseguir els mateixos resultats en quant a la millor distància possible, però amb una millora en el temps d'execució gràcies a l'aprofitament de múltiples processadors o nuclis de la CPU. Per dur-ho a terme, farem servir la interfície de programació OpenMP, que ens ofereix un conjunt de directrius i clàusules per indicar al compilador com paral·lelitzar el nostre codi de manera eficient.

2 Codi seqüencial

Inicialització de dades

Es defineixen les coordenades X i Y de, per exemple, les ciutats i es calcula una matriu de distàncies entre totes les parelles de ciutats utilitzant la fórmula de la distància euclidiana entre dos punts:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
for(i=0; i<nn; i++) X[i]=rand()%(nn*10);
for(i=0; i<nn; i++) Y[i]=rand()%(nn*10);

for(i=0; i<nn; i++) distancia[i][i]=0;

for(i=0; i<nn; i++)
    for(j=i+1; j<nn; j++)
        distancia[i][j]= distancia[j][i] = sqrt(pow(X[i]-X[j],2) + pow(Y[i]-Y[j],2));
```

Loop principal (Algorisme greedy)

L'algorisme selecciona cada ciutat com a punt de partida. Després, es construeix el recorregut des del punt de partida seleccionat. En cada iteració del segon bucle, es busca la ciutat més propera a la ciutat actual (que encara no ha estat visitada). Aquesta ciutat és afegida al recorregut i la distància entre les ciutats s'afegeix a la distància total del recorregut.

Aquest procés es repeteix fins que s'han visitat totes les ciutats. Durant aquest procés, es fa una PODA. Consisteix en si la distància parcial del recorregut supera la millor distància trobada fins al moment, el procés es deté, ja que no pot trobar una solució millor.

```
millor = GRAN;

for (primer=0; primer<nn; primer++) {

     dist = 0;
     for(i=0;i<nn;i++) cami[i]=-1;
     cami[primer]=0;
     actual = primer;


    for (i=1; i<nn; i++) {
         dmin = GRAN;
         index=0; // redundant
         for (j=0; j<nn; j++) {
             if (cami[j]==-1 && actual!=j && distancia[actual][j] < dmin) {
                 dmin = distancia[actual][j];
                 index= j;
             }
         }
         actual = index;
         cami[actual] = i;
         dist += dmin;
         // PODA
         if (dist >= millor) { dist = 0; break;}
     }
     if (dist) {
         dmin = distancia[actual][primer];
         dist += dmin;
         if (dist < millor) {
             for(i=0;i<nn;i++) bo[cami[i]]=i;
             millor = dist;
         }
         distancia[primer][nn]=dist;  // per guardar alternatives
     }
 }
```

Actualització de la millor solució

Si es troba el recorregut més curt que el millor recorregut trobat fins al moment, aquest nou recorregut es guarda com a la millor solució.

```
     // codi repetit: pertany al loop principal
     if (dist) {
         dmin = distancia[actual][primer];
         dist += dmin;
         if (dist < millor) {
             for(i=0;i<nn;i++) bo[cami[i]]=i;
             millor = dist;
         }
         distancia[primer][nn]=dist;  // per guardar alternatives
     }
```

Impressió de la solució

Al final del procés, es mostra per pantalla la millor solució trobada, així com la distància total recorreguda en el recorregut.

```
     printf("Solucio :\n");
     for(i=0; i<nn; i++) printf("%d\n",bo[i]);
     printf ("Distancia %g == %g\n",millor,distancia[bo[0]][nn]);

     exit(0);
```

3 Anàlisis de les dependències de dades

Dependències de dades

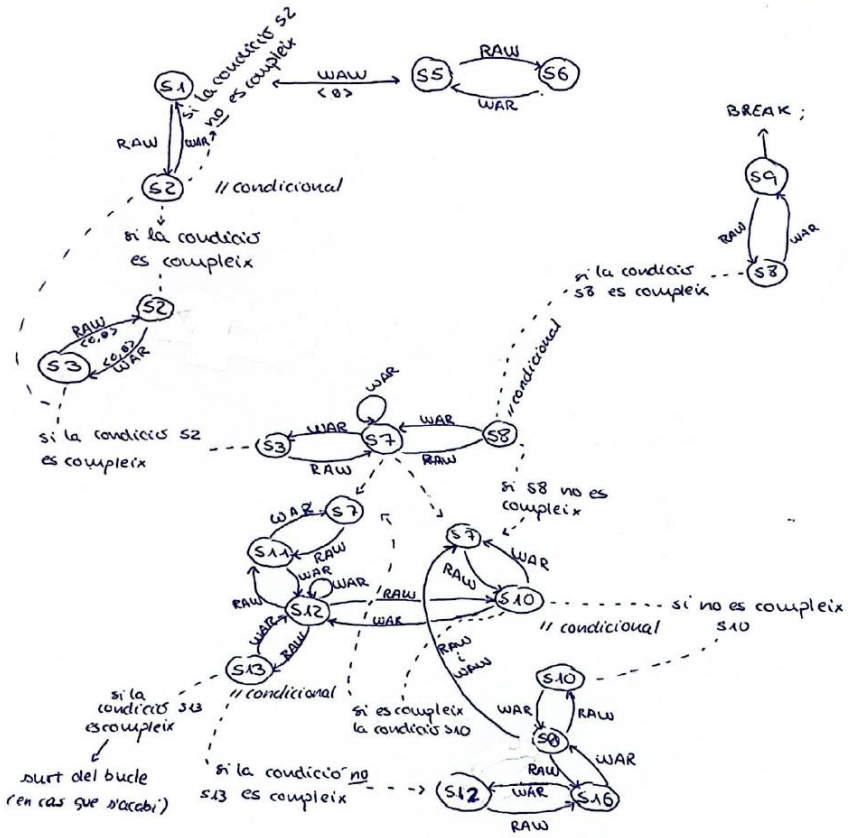
```
#pragma omp parallel for default (none) private(i, j, actual, dist, index, dmin, cami)
shared(distancia, nn, bo, vect_millor, vect_bo)

    for (primer=0; primer<nn; primer++) {
        dist = 0;   // --- S0
        for(i=0;i<nn;i++) cami[i]=-1;
        cami[primer]=0;
        actual = primer;   // --- S1
        for (i=1; i<nn; i++) {
            dmin = GRAN;
            index=0; // redundant
            for (j=0; j<nn; j++) {
                if (cami[j]==-1 && actual!=j && distancia[actual][j] < dmin) {   // --- S2
                    dmin = distancia[actual][j];   // --- S3
                    index= j;   // --- S4
                }
            }
            actual = index;   // --- S5
            cami[actual] = i;   // --- S6
            dist += dmin;   // --- S7 dist = dist + dmin
            // PODA
            if (dist >= millor) { // --- S8
                dist = 0; // --- S9
                break;
            }
        }
        if (dist) {   // --- S10
            dmin = distancia[actual][primer];   // --- S11
            dist += dmin;   // --- S12 dist = dist + dmin
            if (dist < millor) {   // --- S13
                for(i=0;i<nn;i++)
                    bo[cami[i]=i;   // --- S14
                millor = dist;   // --- S15
            }
            distancia[primer][nn]=dist;  // per guardar alternatives   // --- S16
        }
    }
```

Per fer el graf, primer hem analitzat totes les variables individualment per poder descartar o no les que no tenen dependència. Després, al ser un programa llarg, és complicar visualitzar o realitzar el graf sense una estructura prèvia.

El que em decidit ha sigut dividir el codi per trossos, d'aquesta manera ens quedarien seccions de codi bastant simples de fer. Una vegada fet, hem ajuntats els grafs, tenint en compte que també poden haver dependències entre ells.

Finalment, quan ja hem analitzat el codi amb profunditat i el graf acabat, hem pogut veure el programa amb més perspectiva i hem pogut ser més crítiques amb el graf, corregint així petits errors que hem anat veient.

A la següent pàgina es pot veure el graf de dependències de les dades d'aquest bucle, indicant el seu tipus (WAR, RAW, WAW) i nivell.



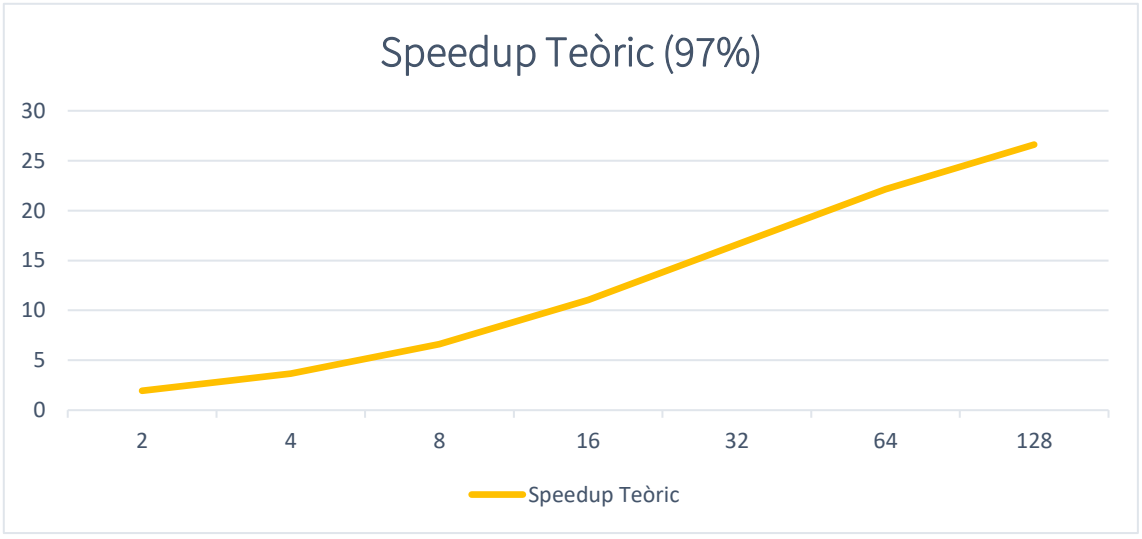
4 Codi paral·lel

En aquesta secció, explicarem com hem determinar quines parts del codi es poden paral·lelitzar i quines han de romandre seqüencials.

Per poder paral·lelitzar el codi seqüencial, haurem d'aplicar la llei d'Amdahl. Aquesta llei estableix el màxim teòric en la millora del rendiment d'un programa quan es paral·lelitzar només una part d'aquest.

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

On P és la proporció del programa que es pot paral·lelitzar i N és el nombre de processadors o nuclis disponibles. Per fer una estimació del SpeedUp que podem esperar, calcularem el SpeedUp teòric suposant que la fracció paral·lelitzables és $\sim 97\%$:



4.1 Propostes de paral·lelització

Inicialment, vam plantejar dues alternatives per paral·lelitzar el codi; La primera consistia en utilitzar la directiva *reduction* de OpenMP, ja que això ens permetia que cada thread tingués una còpia pròpia de les variables `millor` i `bo`.

Tot i això, degut al funcionament del *reduction*, això no ho podíem fer directament: podíem fer la reducció per la variable `millor`, però perdíem el contingut de la variable `bo`.

Per solucionar aquest problema, vam fer la reducció manualment, convertint cada variable en un vector nou que desés el resultat de cada *thread*, per després fer un bucle addicional per determinar el mínim global de tots els threads.

Encara que aquesta implementació funcionava i donava un bon speedup, al Teen vam obtenir una mitja harmònica de 2,85, que era inferior al demanat. És per això, que vam descartar aquesta proposta i ens vam centrar en la que explicarem a continuació; consisteix en no utilitzar la directiva *reduction* i, en el seu lloc, crear una secció crítica per al condicional on es modifiquen les variables millor i bo.

4.2 Explicació de la proposta final de paral·lelització

Llibreries necessàries

La llibreria *omp.h* és l'encapçalament necessària per utilitzar OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <omp.h>   // Inclou l'encapçalament d'OpenMP
```

Paral·lelització del càlcul de distàncies

Apliquem la directiva `parallel for` per paral·lelitzar el càlcul de les distàncies entre els punts. S'indica a OpenMp que volem paral·lelitzar el bucle exterior. Cada iteració d'aquest bucle es distribueix entre els fils d'execució disponibles.

L'ús de `private(j)` indica que la variable `j` és privada per a cada fil d'execució. Això significa que cada fil d'execució tindrà una còpia pròpia de la variable `j`, evitant problemes de condició de carrera.

```
#pragma omp parallel for private(j)
    for (i=0; i<nn; i++){
        distancia[i][i]=0;
        for(j=i+1; j<nn; j++){
            distancia[i][j]= distancia[j][i] = sqrt(pow(X[i]-X[j],2) + pow(Y[i]-Y[j],2));
        }
    }
```

Paral·lelització del bucle principal amb OpenMp

El loop principal de la cerca del camí més curt es paral·lelitza amb la directiva `parallel for`. Dins d'aquest directiva, podem veure que utilitzem les clàusules `private` i `shared`.

La clàusula `private`, especifica les variables locals a cada fil d'execució, mentre que la clàusula `shared` especifica les variables que es comparteixen entre els fils d'execució.

Així doncs, amb la clàusula `private(i, j, actual, dist, index, dmin, camí)` s'estableix que cada fil d'execució tindrà les seves pròpies còpies d'aquestes variables, mentre quela resta de variables es compartiran entre tots els fils d'execució.

Com que les variables `millor` i `bo` es comparteixen entre tots els fils, hem de controlar la seva modificació per garantir que sempre continguin el valor adequat i que no hi hagi problemes de concurrència. Per fer aquesta sincronització afegim una directiva pragma de secció crítica, `#pragma omp critical,` que ens permet garantir que només un fil pugui accedir alhora a aquest condicional.

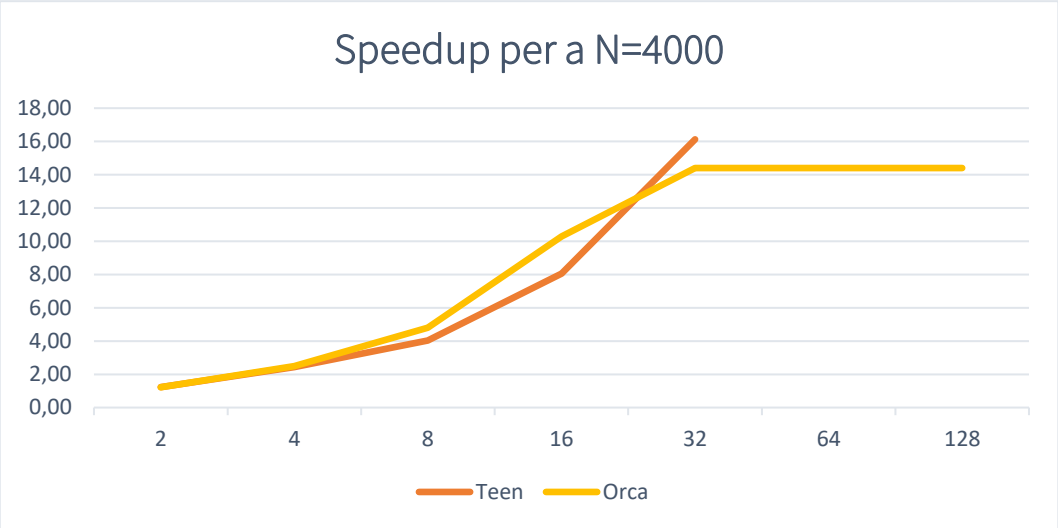
```
#pragma omp parallel for private(i, j, actual, dist, index, dmin, cami)

    for (primer=0; primer<nn; primer++) {
        dist = 0;
        for(i=0;i<nn;i++) cami[i]=-1;
        cami[primer]=0;
        actual = primer;
        for (i=1; i<nn; i++) {
            dmin = GRAN;
            index=0;
            for (j=0; j<nn; j++) {
                if (cami[j]==-1 && actual!=j && distancia[actual][j] < dmin) {
                    dmin = distancia[actual][j];
                    index= j;
                }
            }
            actual = index;
            cami[actual] = i;
            dist += dmin;
            // PODA
            if (dist >= millor) { dist = 0; break;}
        }
        if (dist) {
            dmin = distancia[actual][primer];
            dist += dmin;

            #pragma omp critical
            {
                if (dist < millor){
                    for(i=0;i<nn;i++) bo[cami[i]]=i;
                    millor = dist;
                }
            }
            distancia[primer][nn]=dist;
        }
    }
```

5 Resultats

Un cop vam comprovar que el resultat obtingut amb el codi paral·lel era idèntic al del codi seqüencial, $2{,}22753x10^6$ per a $N = 4000$, vam dur a terme l'estudi de la millora del temps d'execució i el speedup per a les dues màquines, Teen i Orca, segons la quantitat de nuclis assignats per a l'execució del codi.

5.1 Temps d'execució

Els resultats obtinguts per al temps d'execució van ser els següents:



Podem observar com, a mida que augmentem els nuclis disponibles per a l'execució del codi paral·lel, el temps d'execució baixa de forma substancial.

Això es deu a que OpenMP aprofita al màxim aquests nuclis per tal de dividir la tasca en la major quantitat de *threads* possible, el que disminueix el temps de càlcul gràcies a la seva operació en paral·lel.

Tot i això, aquesta millora no és infinita; arriba un punt en el que encara que dividim la feina en més *threads*, el temps d'execució gairebé no es veu afectat. Això es deu al fet que arriba un punt on dividir més la feina ja no suposa un avantatge degut a que cada thread ja triga molt poc en fer la seva feina assignada.

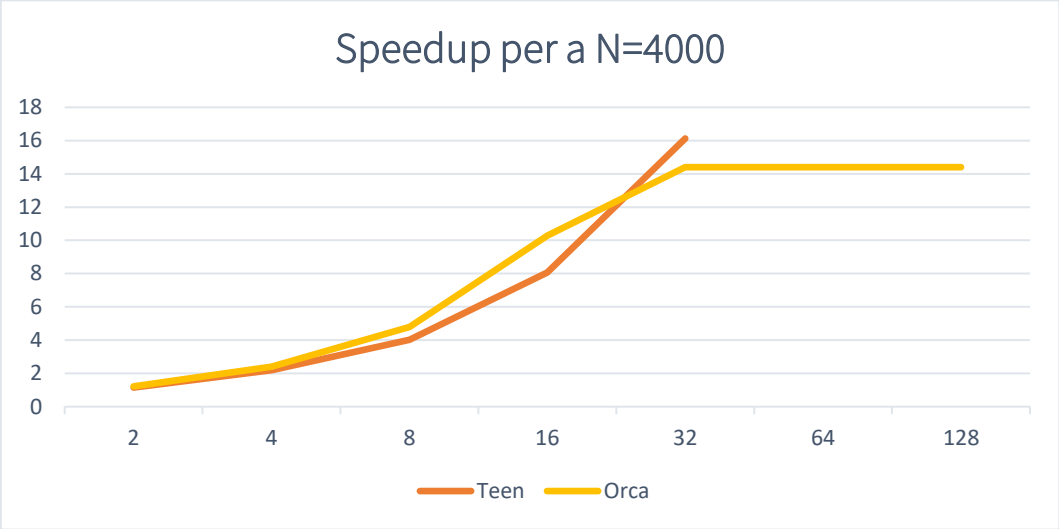
En aquesta situació, afegir més *threads* ja no suposa un avantatge, sinó al contrari, ja que estem afegint més *overhead* per totes les tasques necessàries per al a seva creació i gestió. Tot això es pot observar clarament en l'execució en Orca, ja que a partir de 32 nuclis, la millora en el temps d'execució ja és gairebé imperceptible.

5.2 SpeedUp

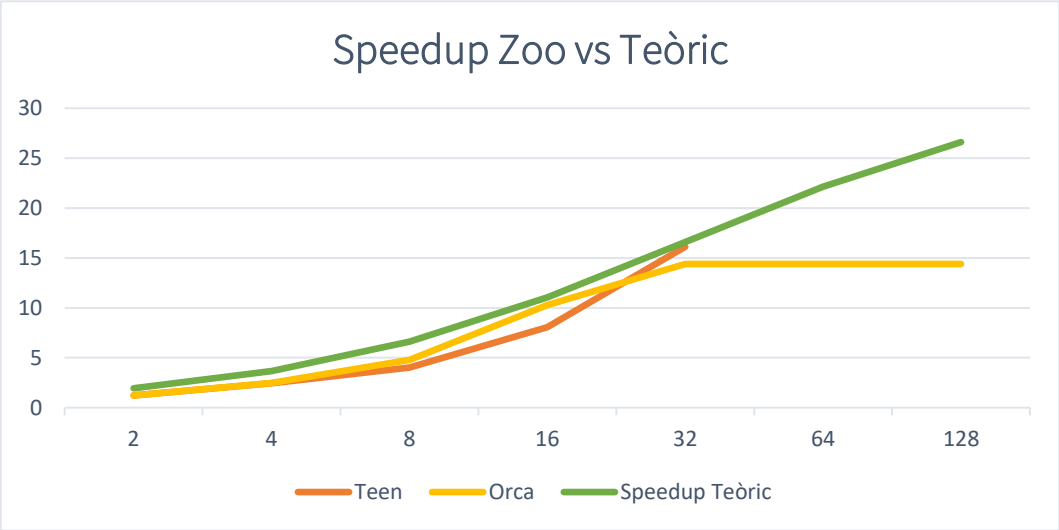
Un cop tenim el temps d'execució, podem calcular el SpeedUp amb la fórmula:

$$Speedup = \frac{T_{seq}}{T_{par}}$$

Els resultats obtinguts es mostren en el gràfic a continuació:



Si comparem aquest resultat amb el SpeedUp teòric obtingut anteriorment, veiem que ens apropem bastant. Evidentment, assolir el SpeedUp teòric no és factible degut a que la llei d'Amdahl no té en compte d'overhead creat per tots els processos de creació de threads.



De forma similar al que passava amb el temps d'execució, veiem com a mida que augmentem els nuclis, el SpeedUp respecte a la versió seqüencial també augmenta. A més, també veiem com en el cas d'Orca, a partir dels 32 threads es produeix una aturada d'aquesta millora, ja que hem arribat al màxim de la millora que es pot aconseguir afegint threads.

La mitjana harmònica del SpeedUp per a Teen i Orca és la següent:

Cores	2	4	8	16	32	64	128	Mitjana Harmònica
Teen	1,23	2,43	4,03	8,06	16,13			3,014
Orca	1,22	2,48	4,80	10,29	14,40	14,40	14,40	4,032

6 Pseudocodi del programa en paral·lel proposat

```
Funció principal(na, arg)
    Enter na
    Array de cadenes arg[]
    Enter nn, i, j, primer, actual, index
    Flotant dmin, dist, millor

    Assegurar(na == 2)
    Escriure "Dimensio", arg[1]
    nn = convertir_a_enter(arg[1])
    Assegurar(nn <= N)

    // Generar coordenades aleatòries per X i Y
    Per cada i de 0 a nn fer
        X[i] := rand() % (nn * 10)
    Final del bucle

    Per cada i de 0 a nn fer
        Y[i] := rand() % (nn * 10)
    Final del bucle

    // Calcular les distàncies entre els punts
    Per cada i de 0 a nn fer en paral·lel
        distancia[i][i] = 0
        Per cada j de i+1 a nn fer
            distancia[i][j] = distancia[j][i] = sqrt((X[i] - X[j])^2 + (Y[i] - Y[j])^2)
        Final del bucle
    Final del bucle

    // Iteració per a tots els punts com a punt de partida
    millor = GRAN

    Per cada primer de 0 a nn fer en paral·lel
        dist = 0
        Per cada i de 0 a nn fer
            cami[i] = -1
        Final del bucle

        cami[primer] = 0
        actual = primer

        Per cada i de 1 a nn fer
            dmin = GRAN
            index = 0
            Per cada j de 0 a nn fer
                Si (cami[j] == -1 i actual ≠ j i distancia[actual][j] < dmin) llavors
                    dmin = distancia[actual][j]
                    index = j
                Fi de si
            Final del bucle

            actual = index
            cami[actual] = i
            dist = dist + dmin

            // PODA
            Si (dist ≥ millor) llavors
                dist = 0
                Sortir del bucle
            Fi de si
        Final del bucle
```

```
        Si (dist ≠ 0) llavors
           dmin = distancia[actual][primer]
           dist = dist + dmin
           // Secció crítica per actualitzar la millor solució
           Secció crítica
               Si (dist < millor) llavors
                   Per cada i de 0 a nn fer
                       bo[cami[i]] = i
                   Final del bucle
                   millor = dist
               Fi de si
           Fi de la secció crítica
           distancia[primer][nn] = dist // per desar alternatives
       Fi de si
   Final del bucle

   // Mostrar la solució i la distància mínima
   Escriure "Solucio :"
   Per cada i de 0 a nn fer
       Escriure bo[i]
   Final del bucle
   Escriure "Distancia", millor, "==", distancia[bo[0]][nn]

   Sortir(0)
Fi de la funció principal
```

7 Conclusions

Amb el codi paral·lel proposat hem pogut complir amb l'objectiu del a pràctica, que consistia en augmentar la velocitat d'execució del programa seqüencial (speedup) sense modificar el seu comportament. Per fer això hem utilitzat les directives de OpenMP, que han permès paral·lelitzar els bucles del codi de forma molt eficient.

A més, també hem pogut comprovar les limitacions de certes directives, com és el cas del *reduction*. Degut a que necessitàvem obtenir el camí mínim a més de la distància mínima, no hem pogut utilitzar la directiva de reducció, ja que només permet treballar amb una variable. Fent la reducció manual, encara que el codi funcionava, no arribava al speedup demanat, pel que hem acabat utilitzant una secció crítica.

Tot això ens ha permès veure la utilitat de l'eina OpenMP per a la paral·lelització de bucles, però també per entendre les seves limitacions i saber buscar alternatives per obtenir els resultats desitjats.