Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

Departament d'Enginyeria Informàtica i Matemàtiques
[DEIM]
Universitat Rovira i Virgili

Computació paral·lela i massiva:

Pràctica 1.1:

Algorisme d'agregacions Kmean.

Estudiants: Leandro Favio Gomez Racca

Miriam Gertrudix Pedrola

Professor: Carles Aliagas Castell

Índex:

Enunciat

Algorisme d'agregacions Kmean. A partir d'un vector inicialitzat amb dades aleatòries, aplicar l'algorisme per obtenir 200 agrupacions dels 600.000 elements del vector.

Temps d'execució seqüencial amb les opcions "-O3" per N=600.000 i 200 centroides:

a Gat -> 65.15 segons,

a Roquer -> 21.25 segons

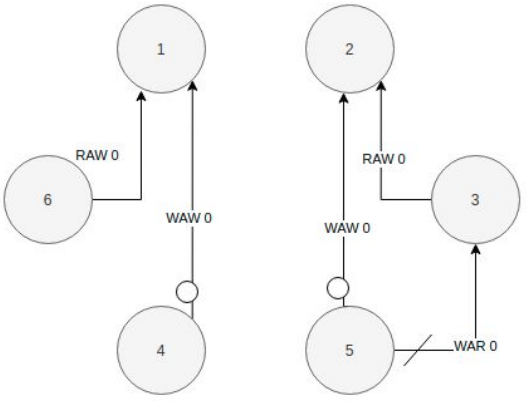
- Es programarà amb OpenMP
- Speedup mínim a GAT: 3
- La versió paral·lela ha de donar el mateix resultat que la seqüencial. (Cal comprovar els valors de sortida)
- Cal executar el codi, forçant la creació de threads de 2 fins a 16.

Anàlisi de les dependències:

Per a l'anàlisi de les dependències ens centrarem amb els diferents bucles amb possibilitat de paral·lelitzar.

Comencem per el primer bucle, En aquest bucle veiem que les principals dependències son causades per les variables dif i min. Totes les dependències son de nivell 1 això fa que no tingui cap mena de problema a la hora de paral·lelitzar el bucle. Tot i així haurem de vigilar el comportament d'aquestes variables perquè no ens afecti al resultat.

```
per  i := 0 , i < Ndades, i := i+1, fer
S1:       min := 0
S2:       dif := abs( fV[i] - fR[0] )
          per j := 1, j < fK, j := j+1, fer
S3:                 si abs( fV[i] -fR[j] ) < dif llavors
S4:                           min := j
S5:                           dif := abs( fV[i] - fR[j] )
                    fsi
          fper
S6:       fD[i] := min
fper
```



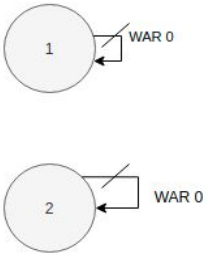
Aquest for no conté cap dependencia, ja que simplement posa els dos vectors a 0.

```
per i := 0, i < Kagrupacions, i := i+1 fer
S1:       fS[i] := 0;
S2:       fA[i] := 0;
fper
```

Els dos fors anteriors tracten dades diferents, Això vol dir que son independents i que per tant es poden executar a la vegada.

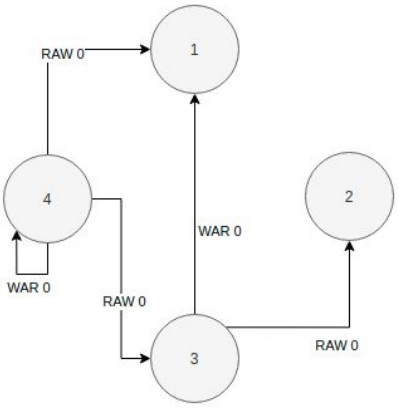
En aquest for les dependències son a si mateixos i WAR i per tant l'execució ha de ser seqüencial o, per altra banda, realitzar-ho de
forma atòmica o mitjançant semàfors.

```
per  i := 0 , i < Ndades, i := i+1, fer
S1:       fS[fD[i]] := fS[fD[i]] + fV[i];
S2:       fA[fD[i]] := fA[fD[i]] + 1;
fper
```



En aquest bucle per ens passa semblant que en el primer, les dependències que més ens poden preocupar son les que creen les variables t i la variable dif. En aquest cas la variable dif es tracta d'un acumulatiu. Per tant serà possible paral·lelitzar sempre i quan dif ens doni el resultat pertinent un cop acabada l'execucció.

```
per i := 0, i < Kagrupacions, i := i + 1, fer
S1:     t := fR[i]
S2:     si fA[i] llavors
S3:             fR[i] := fS[i] / fA[i]
        fsi
S4:     dif := dif + abs(t - fR[i])
fper
```



[SENSE DEPENDENCIES]

```
per i := 0, i < N, i := i + 1, fer
s1:     V[i] = (aleatori mod aleatori) / N;
fper
```

[SENSE DEPENDENCIES]

```
per i := 0, i < G, i := i + 1, fer
S1:     R[i] = V[i];
fper
```

proposta:

Després de estudiar les diferents seccions del codi, i d'analitzar el comportament del seus bucles vam prendre les següents decisions:

En la part del main se'ns complicava intentar paral·lelitzar els bucles perquè el tractament amb aleatoris feia que el resultat no sigués el correcte així que vam decidir deixar-lo en seqüencial.

Després ens vam fixar en la part del quicksort, i despres de buscar més informació, vam arribar a la conclusió que era un algorisme molt optimitzat i rapid i que per tant ens resultaria molt complicat de paral·lelitzar-lo bé i de forma balancejada. D'aquesta manera vam decidir que la millor secció de codi per a paral·lelitzar era la de la funció Kmean.

En el primer en que ens vam basar va ser en el primer for bidimensional. Un cop analitzades les dependències i al ser fins a N vam decidir que el podíem paral·lelitzar el for exterior per a disminuir considerablement el temps d'execució del codi.
Després d'aquesta millora, ens vam adonar que no tenia cap dependència amb el for que hi havia a continuació i que per tant es podien realitzar els dos fors a la vegada. La nostra sorpresa va ser que si intentavem paralelitzar amb un pragma parallel for el segon for aquest tardava més. Després d'estudiar aquest comportament vam entendre que això era degut a que hi havia poc balanceig entre els threads. Així que la nostra decisió final va ser executar el bucle bidimensional i el següent en paral·lel, i el bucle bidimensional dividir el treball per als diferents threads. les variables min i dif vam decidir privatitzar-les per a cada thread per a evitar conflictes.

Després d'intentar paral·lelitzar el tercer for no vam aconseguir cap millora en quant a temps d'execució. Així que vam decidir deixar-lo en seqüencial.

I per últim, vam decidir paral·lelitzar l'últim for. Per a fer-ho vam decidir privatitzar la variable t per el mateix motiu que amb el dif i el min anteriors, son variables les quals només ens importa el valor durant la iteració del bucle i és important que els threads no s'estiguin sobreposant les dades uns als altres. Per a la variable dif en aquest cas és diferent ja que es tracta d'un acumulatiu i que el do while exterior depèn del valor d'aquesta. Així que vam decidir utilitzar un reduction que sumes el dif final de tots els threads.

Pseudocodi:

```
funció kmean (enter fN, enter fK, long fV[], long fR[], enter fA[])

enter i,j,min,iter=0;
long dif,t;
long fS[G];
enter fD[N];

        fer
                #pragma omp parallel
                {
                        #pragma omp for private (min, dif)
                        per  i := 0 , i < Ndades, i := i+1, fer
                                min := 0
                                dif := abs( fV[i] - fR[0] )
                                per j := 1, j < fK, j := j+1, fer
                                        si abs( fV[i] -fR[j] ) < dif llavors
                                                min := j
                                                dif := abs( fV[i] - fR[j] )
                                        fsi
                                fper
                                fD[i] := min
                        fper

                        per i := 0, i < Kagrupacions, i := i+1 fer
                                fS[i] := 0;
                                fA[i] := 0;
                        fper
                }

                per  i := 0 , i < Ndades, i := i+1, fer
                        fS[fD[i]] := fS[fD[i]] + fV[i];
                        fA[fD[i]] := fA[fD[i]] + 1;
                fper

                dif := 0;

                #pragma omp parallel for reduction(+:dif) private (t)
                per i := 0, i < Kagrupacions, i := i + 1, fer
                        t := fR[i]
                        si fA[i] llavors
                                fR[i] := fS[i] / fA[i]
                        fsi
                        dif := dif + abs(t - fR[i])
                fper
```

```
                iter := iter + 1;
        mentre (dif);

        Escriure("iter:",iter);
ffunció

funcio qs(int ii, int fi, long fV[], int fA[])
{
        enter i,f,j;
        long pi,pa,vtmp,vta,vfi,vfa;

        pi := fV[ii];
        pa := fA[ii];
        i := ii +1;
        f := fi;
        vtmp := fV[i];
        vta := fA[i];

        mentre  (i <= f) fer
                si (vtmp < pi) llavors
                        fV[i-1] := vtmp;
                        fA[i-1] := vta;
                        i := i + 1;
                        vtmp := fV[i];
                        vta := fA[i];
                sino
                        vfi := fV[f];
                        vfa := fA[f];
                        fV[f] := vtmp;
                        fA[f] := vta;
                        f := f - 1;
                        vtmp := vfi;
                        vta := vfa;
                fsi
        fmentre
        fV[i-1] := pi;
        fA[i-1] := pa;

        si (ii < f) llavors
                qs(ii,f,fV,fA);
        fsi
        si (i < fi) llavors
                qs(i,fi,fV,fA);
        fsi
ffunció
```

```
funció principal
	int i;

	per i := 0, i < N, i := i + 1, fer
		V[i] = (aleatori mod aleatori) / N;
	fper

	// primers candidats
	per i := 0, i < G, i := i + 1, fer
		R[i] = V[i];
	fper

	// calcular els G mes representatius
	kmean(N,G,V,R,A);

	qs(0,G-1,R,A);

	per i := 0, i < G, i := i + 1, fer
		Escriure("R[i] : R[i] te A[i] agrupats");
	fper
ffunció
```

Codi:

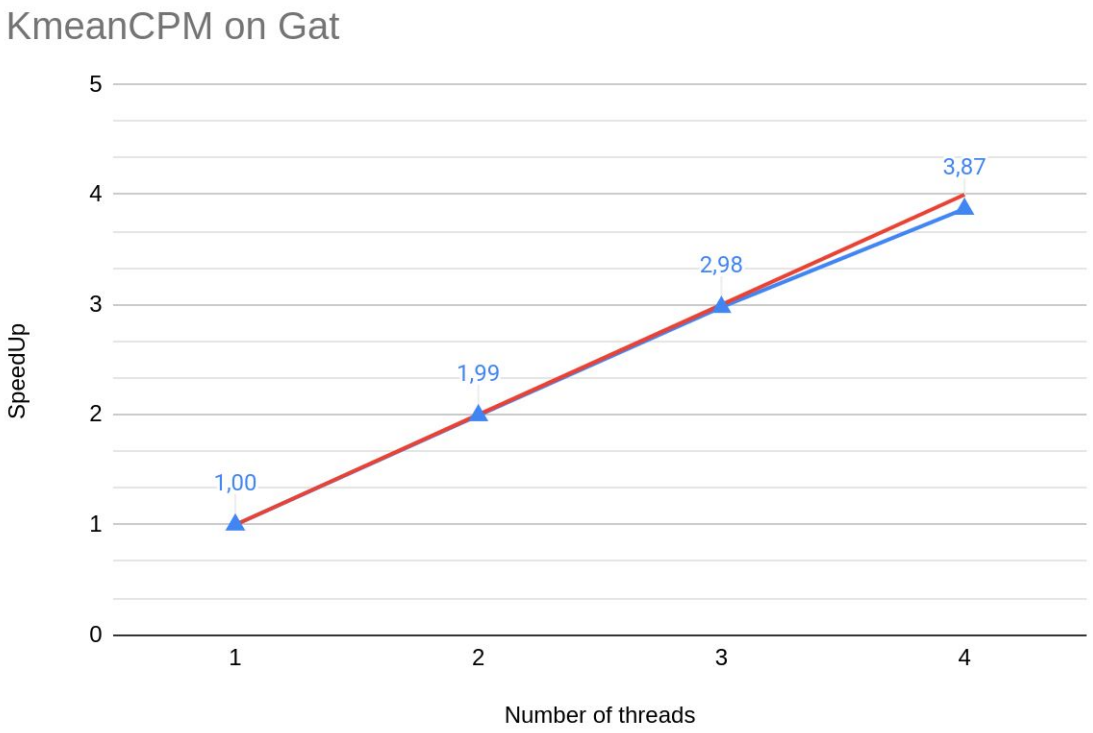
```
#include <stdlib.h>
#include <stdio.h>

#define N 600000
#define G 200

long V[N];
long R[G];
int A[G];

void kmean(int fN, int fK, long fV[], long fR[], int fA[])
{
int i,j,min,iter=0;
long dif,t;
long fS[G];
int fD[N];

   do
   {
       #pragma omp parallel
       {
           #pragma omp for private (min, dif)
           for (i=0;i<fN; i++)
           {
               min = 0;
               dif = abs(fV[i] -fR[0]);
               for (j=1;j<fK;j++)
               {
                   if (abs(fV[i] -fR[j]) < dif)
                   {
                       min = j;
                       dif = abs(fV[i] -fR[j]);
                   }
               }
               fD[i] = min;
           }

           for(i=0;i<fK;i++) fS[i] = fA[i] = 0;
       }
```

```
        for(i=0;i<fN;i++)
        {
            fS[fD[i]] += fV[i];
            fA[fD[i]] ++;
        }

        dif = 0;

        #pragma omp parallel for reduction(+:dif) private (t)
        for(i=0;i<fK;i++)
        {
            t = fR[i];
            if (fA[i]) fR[i] = fS[i]/fA[i];
            dif += abs(t - fR[i]);
        }

        iter ++;
    } while(dif);

    printf("iter %d\n",iter);
}

void qs(int ii, int fi, long fV[], int fA[])
{
    int i,f,j;
    long pi,pa,vtmp,vta,vfi,vfa;

    pi = fV[ii];
    pa = fA[ii];
    i = ii +1;
    f = fi;
    vtmp = fV[i];
    vta = fA[i];

    for (;i <= f;)
    {
        if (vtmp < pi) {
        fV[i-1] = vtmp;
        fA[i-1] = vta;
        i ++;
        vtmp = fV[i];
```

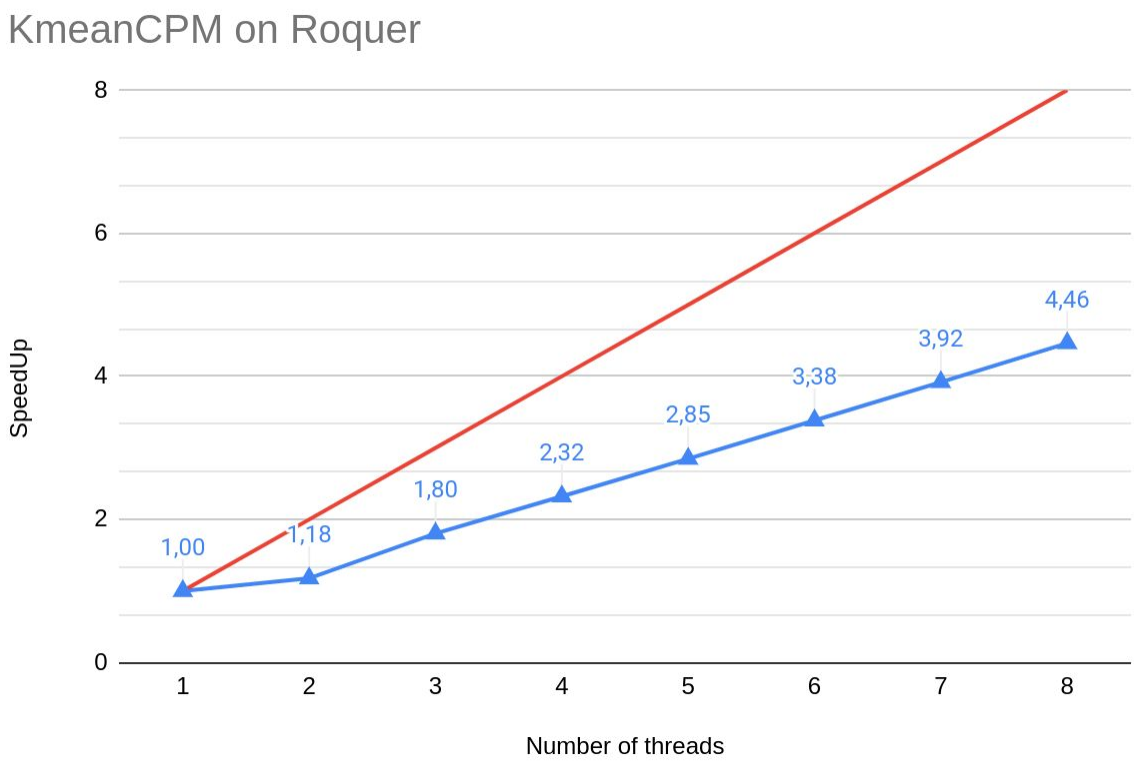
```
        vta = fA[i];
        }
        else {
        vfi = fV[f];
        vfa = fA[f];
        fV[f] = vtmp;
        fA[f] = vta;
        f --;
        vtmp = vfi;
        vta = vfa;
        }
    }
    fV[i-1] = pi;
    fA[i-1] = pa;

    if (ii < f) qs(ii,f,fV,fA);
    if (i < fi) qs(i,fi,fV,fA);
}

int main()
{
    int i;

    for (i=0;i<N;i++) V[i] = (rand()%rand())/N;

    // primers candidats
    for (i=0;i<G;i++) R[i] = V[i];

    // calcular els G mes representatius
    kmean(N,G,V,R,A);

    qs(0,G-1,R,A);

    for (i=0;i<G;i++)
    printf("R[%d] : %ld te %d agrupats\n",i,R[i],A[i]);

    return(0);
}
```

Speed Up:

A continuació analitzem l'Speed up del nostre codi paral·lelitzat per a N=600.000 i 200 centroides



En aquest gràfic veiem en color vermell l'Speed Up teoric segons la llei d'Amdahl i en color blau els nostres resultats obtinguts. Com es pot apreciar en el Gat els resultats son molt similars.



En Canvi amb el Roquer veiem que s'allunya bastant el nostre codi del Speed Up teòric. Tenim un speed up similar al Gat tot i que esperàvem doblar aquest speed up ja que en el roquer cada core te dos threads i creiem que aixo és degut al hyperthreading.