PRÁCTICA 1.2
Computación Paralela y Masiva

Marcos Esteve Hernández

Geovanny Alexander Risco Carrera

Indice

Enunciado de la práctica

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

El codi és el mateix que la practica P1.1.
El temps seqüencial a les màquines POP és de 64.5 segons.

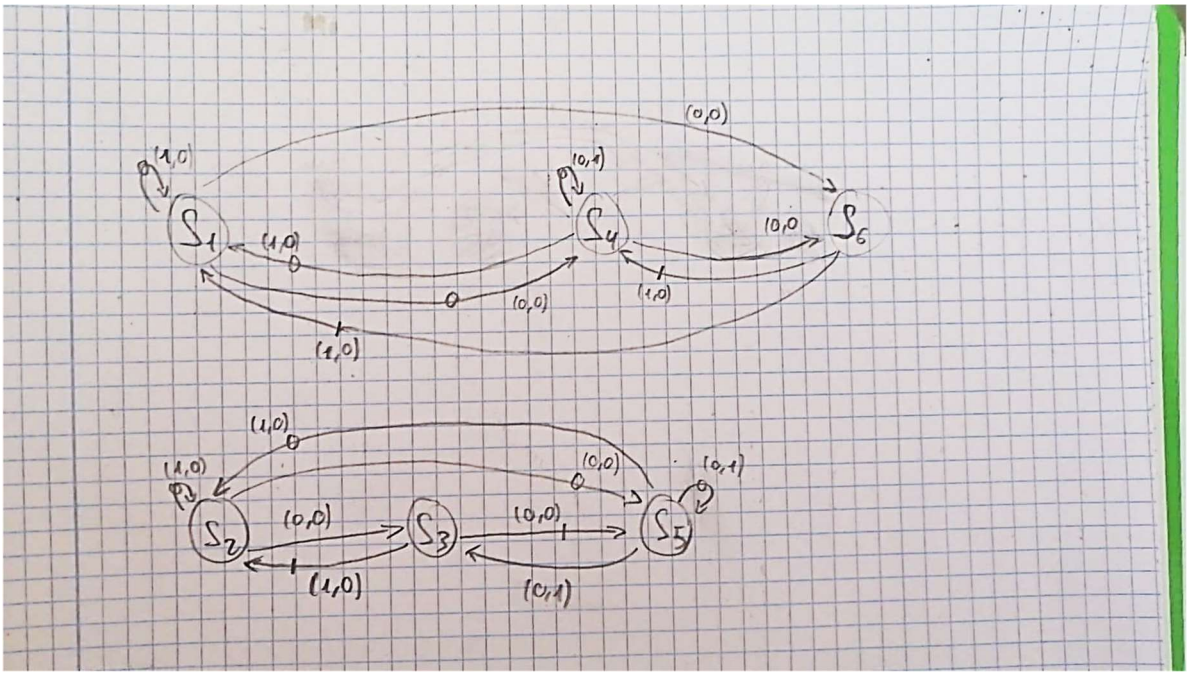
- Es programarà amb MPI
- L'execució es farà per els nombre de processos que indica la taula. Així son 11 execucions.
- S'ha de respectar les dades que surten per la sortida estàndard i en la versió paral·lela ha de donar el mateix resultat.
- El vector resultant ha de quedar emmagatzemat sencer al procés 0.
- L'speedup en qualsevol configuració ha de ser superior a 1.5
- L'speedup harmònic a les màquines POP ha de ser superior a 3.0

Análisis de dependencia de datos

A continuación, detallaremos los gráficos de dependencias de los bucles que aparecen en esta práctica.

El primer bucle de la función kmean estaría representado por el siguiente grafo:

```
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;
       }
```



Existen dependencias que van desde las sentencias correspondientes al bucle i hacia sentencias correspondientes al bucle j, por lo que el bucle interno no se puede paralelizar. Por otro lado, todas las dependencias son de distancia 0, por lo que no existe problema al paralelizar el bucle externo.

El segundo bucle de la función kmean:

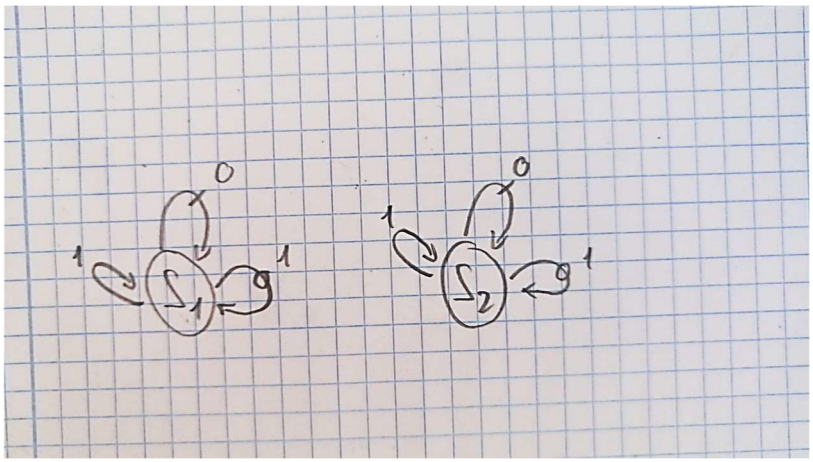
```
        for (i = 0; i < fK; i++)
            fS[i] = fA[i] = 0;
```

Este bucle no tiene dependencias ya que se escribe en direcciones de memoria totalmente diferentes. Por lo tanto, se puede paralelizar.

El tercer bucle de la función kmean:

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

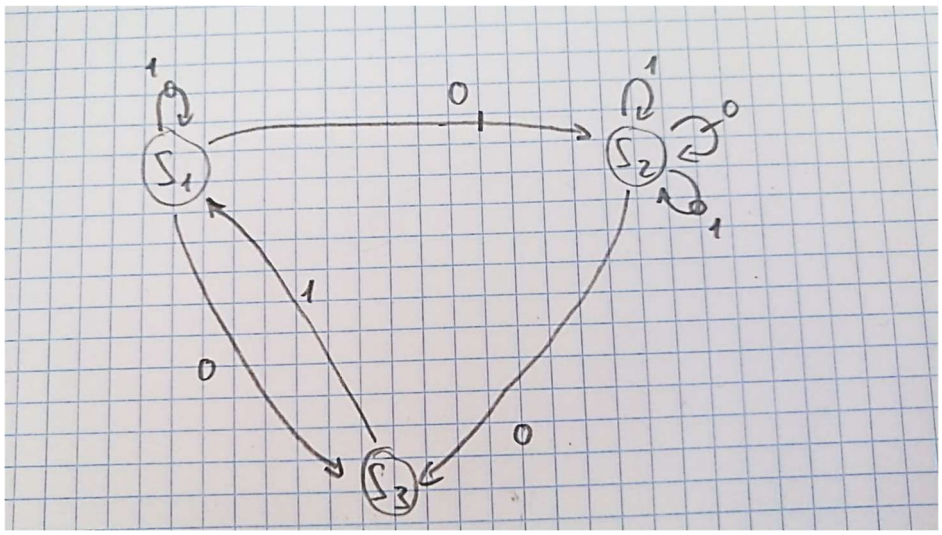
Grafo de dependencias para este bucle:



Este bucle presenta únicamente dependencias internas de distancia 0, por lo que es paralelizable.

Las dependencias del último bucle de la función kmean estarían representadas por el siguiente grafo:

```
for(i=0;i<fK;i++)
       {
        t = fR[i];
        if (fA[i]) fR[i] = fS[i]/fA[i];
        dif += abs(t - fR[i]);
       }
```



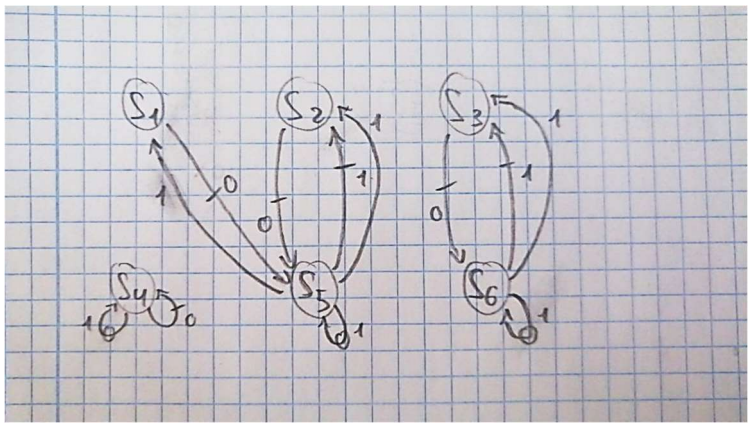
En este bucle existen dependencias internas de distancia 0, pero también dependencias de distancia 1 entre iteraciones del bucle debido a la presencia de variables escalares. Para resolver esto y poder paralelizarlo, sería necesario aplicar técnicas de eliminación de dependencias como la vectorización.

En cuanto a la función QuickSort:

```
while (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;
    }
   }
```

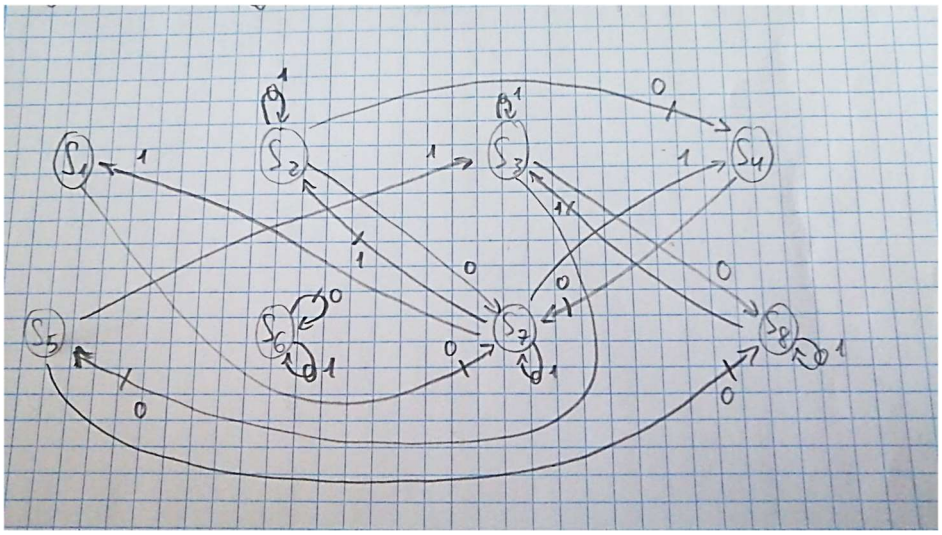
Para generar el grafo de dependencias dividiremos el grafo en 2 partes, una correspondiente al if y otra al else.

Parte del if:



Existen bucles de dependencias de distintas distancias por lo que no es paralelizable.

Parte del else:



Además de esto, debido a que la función Quicksort realiza llamadas recursivas sobre sí misma, estas siguientes ejecuciones también presentan dependencias con las ejecuciones recursivas, ya que los índices de acceso a los vectores como la i o la f coincidirían entre ejecuciones.

En el apartado del main, hay dos bucles correspondientes a inicializaciones pero estos no presentan dependencias internas. El segundo bucle si que depende del primero, por lo que se debe realizar uno despues del otro en secuencial. El primero de los bucles no se puede paralelizar ya que cuenta con la función rand(), la cuál depende internamente de la semilla y está es diferente en cada thread. El segundo bucle sí es paralelizable.

Propuesta de paralelización

La primera propuesta de paralelización que aplicamos es en el main. Solo el proceso root debe inicializar los vectores para que el calculo sea el mismo que en la ejecución secuencial. Es por ello que tanto las inicializaciones como la función Quicksort y el 'print' de los resultados son ejecutadas únicamente por el root, metiendo este código dentro de un if con la condición de ser el root. Sin embargo, como todos los procesos deben tener los vectores para poder ejecutar la función kmean, los dos vectores que ha inicializado el root son enviados a todos los procesos mediante un broadcast.

En esta práctica hemos realizado cambios en la función kmean de la misma manera que en la práctica 1. Únicamente hemos paralelizado el primer bucle de la función kmean, dividiendo el fragmento de vector que tiene que calcular cada thread. En nuestra implementación, todos los procesos tienen el vector entero, pero solo calculan el fragmento que les corresponde. Este fragmento se calcula dividiendo el número total de datos (N) entre el número total de procesos (num_proc). Es por esto que el índice del primer bucle de la función kmean, el que hemos paralelizado, se calcula multiplicando el rango del proceso por el numero de elementos que le tocan a cada proceso. Así, el primero comenzará en el 0 y calculara hasta N/num_proc, el siguiente comenzará en 1*(N/num_proc) hasta 2*(N/num_proc) y así sucesivamente.

Sin embargo, para los siguiente bucles e iteraciones de la función kmean se debe tener el vector completo, por lo que mediante una función allgather hacemos que todos los procesos tengan toda la información. En la implementación, todos los procesos calcularán el resto de bucles de una iteración, lo cual nos beneficia porque así para la siguiente iteración no tendremos que enviar esa información, todos los procesos dispondrán ya de ella.

Cabe comentar el caso concreto en el que el numero total de datos no es divisible entre el numero de procesos. Hemos optado por una implementación en la que el fragmento restante de la división, es decir, el módulo, lo calcula el último proceso. En este caso, cuando hay que reunir los valores calculados por cada proceso, el que ha calculado el fragmento de sobra tiene que enviar más datos que el resto, es decir, el tamaño de datos no es uniforme para todos los procesos. La función allgather no nos permitía enviar tamaños de bufer diferentes. Debido esto, hemos tenido que optar por sustituirla por la función de MPI **Allgatherv**, la cual es un allgather pero que permite especificar el tamaño de datos que envia cada proceso mediante un vector de posiciones y otro de número de elementos.

La función Quicksort es bastante complicada, sobretodo por ser recursiva, además de que solo se ejecuta una vez al final del algoritmo y solo ordena el vector de 200 posiciones, por lo que no vemos relevante su paralelización ya que no supondría un incremento significativo en el speedup global.

Como hemos comentado, la inicialización del vector V no se puede paralelizar por problemas con la función rand(); y probamos paralelizar el bucle de inicialización del vector R, pero el incremento del tiempo de ejecución no era significativo y decidimos no paralelizarlo.

Pseudocódigo de la propuesta

Realizamos el pseudocódigo correspondiente a la paralelización que hemos implementado.

```
def main() {
  if (el_meu_rank== 0){
     #inicializar vectores
  }
   BroadCast(R)
   BroadCast(V)
   kmean()
   if (el_meu_rank==0) {
     quicksort()
   }
}

def kmean() {
   #Calcular trozo del vector que le toca a cada uno
   do {
         for (i=inicio_trozo; i < final_trozo; i++) {
    #Cada thread realizará el bucle en un trozo diferente del vector
         }
        Allgatherv(send_buffer=trozo,size=final_trozo-inicio_trozo, receive_buffer=fD)
        #En este momento todos tendrán la misma fD (vector original)
        #A partir de aquí todos ejecutarán el mismo código con los mismos datos.
    } while(dif)
}
```

Código + paralelización

```
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];
    int elements_per_proc = N/p;
    int ini_proc = el_meu_rank*elements_per_proc;
    int end_proc = (el_meu_rank+1)*elements_per_proc;
    /* En caso de que N no sea divisible por el nº de procesos, asignar e
l resto al ultimo proceso */

    int recvcounts[p],  displs[p];
    for (int x = 0; x < p; x++) {
        recvcounts[x]=elements_per_proc;
        displs[x]=x*elements_per_proc;
    }

    if ((N % p) != 0) {
        if (el_meu_rank==p-1) {
            end_proc = N;
        }
        recvcounts[p-1]=elements_per_proc + N%p;
    }

    int send_buffer_fD[end_proc-ini_proc];

    do
    {
        int cont = 0;
        for (i = ini_proc ; i < end_proc; 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]);
                }
            send_buffer_fD[cont] = min;
            cont++;
        }

        MPI_Allgatherv(send_buffer_fD,end_proc-
ini_proc,MPI_INT, fD, recvcounts, displs, MPI_INT, MPI_COMM_WORLD);
```

```
        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;
        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);

    if (el_meu_rank==0)
        printf("iter %d\n", iter);
}
```

El código de la función kmean. La explicación de la paralelización y nuestras decisiones viene explicada en el apartado de propuesta de paralelización.

```
int main(int num_args, char* args[ ])
{
    int i;

    /* Inicialitzar MPI */
   MPI_Init(&num_args, &args);
    /* Obtenir el rank del proces  */
   MPI_Comm_rank(MPI_COMM_WORLD, &el_meu_rank);
    /* Obtenir el numero total de processos */
   MPI_Comm_size(MPI_COMM_WORLD, &p);

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

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

```
    MPI_Bcast(R, G, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(V, N, MPI_LONG, 0, MPI_COMM_WORLD);

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

    if (el_meu_rank==0){
        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]);
    }

    MPI_Finalize();
    return (0);
}
```

El código de la función main, con los broadcast y los fragmentos de código que ejecuta únicamente el proceso root tal y como se detalla en la propuesta de paralelización.

SpeedUP

Hemos realizado el juego de pruebas con las 11 ejecuciones que se especifican y calculado la media armónica de los speedup, obteniendo un speedup armónico de 3,1688.

Num procesos	Procesos por nodo	Elapsed Time (s)	SpeedUp
2	1	38,65	1,6688
4	2	26,77	2,4094
4	1	24,91	2,5893
8	2	21,93	2,9411
12	3	16,38	3,9377
8	1	18,12	3,5596
16	2	15,86	4,0668
24	3	14,17	4,5519
32	4	13,36	4,8278
64	4	16,40	3,9329
128	4	17,35	3,7176

$$Media\ armónica = \frac{11}{\frac{1}{1{,}6688}+\frac{1}{2{,}4094}+\frac{1}{2{,}5893}+\frac{1}{2{,}9411}+\frac{1}{3{,}9377}+\frac{1}{3{,}5596}+\frac{1}{4{,}0668}+\frac{1}{4{,}5519}+\frac{1}{4{,}8278}+\frac{1}{3{,}9329}+\frac{1}{3{,}7176}} = 3{,}1688$$