PRÁCTICA 1.1
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.

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

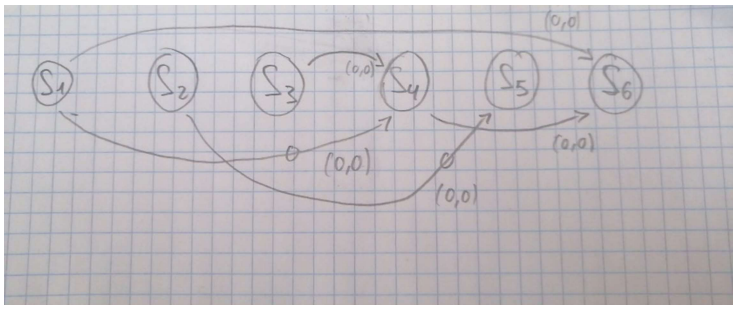
- Es programarà amb OpenMP
- Speedup mínim a GAT: 3
- La versió paral·lela ha de donar el mateix resultat que la seqüencial. (Cal comprovar els valors de sortida)
- Cal executar el codi, forçant la creació de threads de 2 fins a 16.

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:

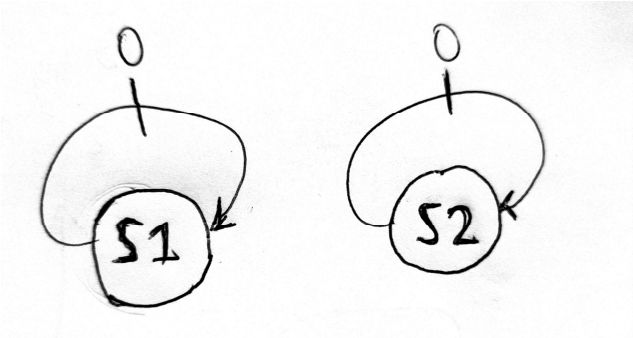
```
        for (i = 0; i < fK; i++)
            fS[i] = fA[i] = 0;
```

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]]++;
        }
```

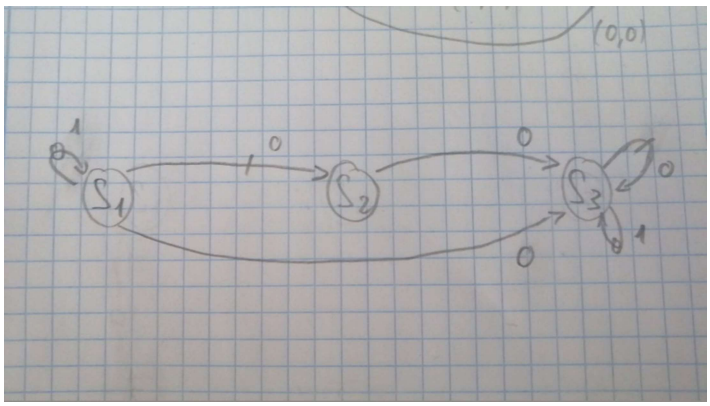
Grafo de dependencias para este bucle:



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]);
       }
```



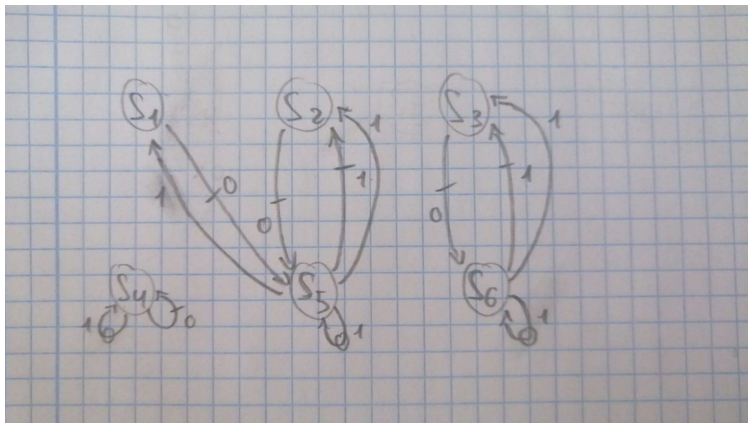
En este bucle existen dependencias internas de distancia 0, pero también dependencias de distancia 1 entre iteraciones del bucle debido a la presencia de variables escalares. Para resolver esto y poder paralelizarlo, sería necesario aplicar técnicas de eliminación de dependencias como la vectorización.

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;
    }
   }
```

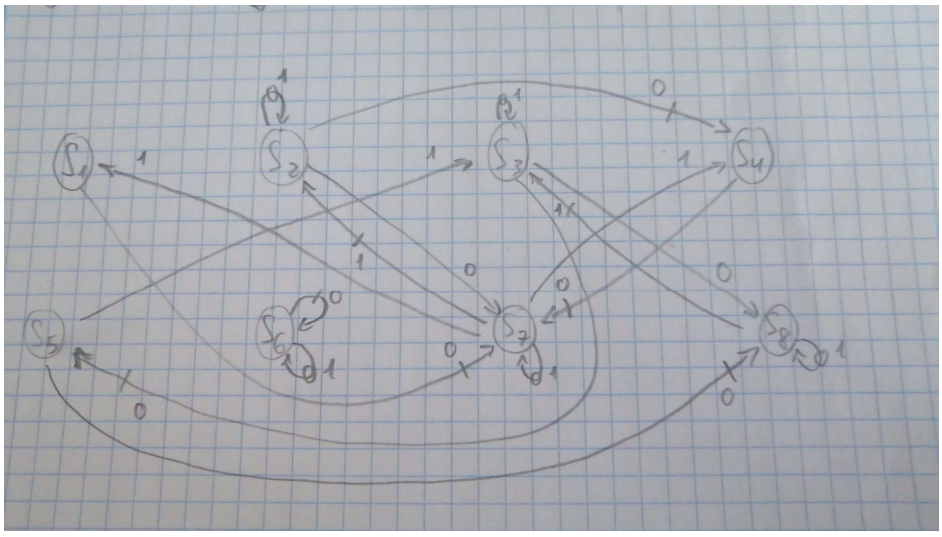
Para generar el grafo de dependencias dividiremos el grafo en 2 partes, una correspondiente al if y otra al else.

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

El primero de los bucles de la función kmean, el cual es un bucle anidado, hemos visto que se puede paralelizar el bucle externo, por lo que realizaremos una paralelización mediante la sentencia pragma omp parallel for con la cláusula Dynamic, para lograr un mejor balance de la carga. Debido a que en el bucle interno encontramos un condicional, en algunas iteraciones la parte de código dentro del condicional se ejecutará y en otras no. Esto puede conllevar un trabajo extra por parte de algún thread. Para compensar esto, utilizamos la cláusula Dynamic para crear un pool de tareas y así el primer thread que acabe (que habrá realizado menos trabajo), comenzará antes con la siguiente tarea, optimizando así el tiempo de ejecución. La variable niter indica el tamaño de cada tarea. Primeramente, se probó con el número por defecto, que es 1, pero no observamos incremento en el speedup notable debido a la sobrecarga que genera estar cambiando o asignando nuevas tareas a cada uno de los threads. Por ello, a continuación, probamos un número mayor, y buscamos un número que fuera múltiplo del numero de cores que utiliza la máquina. Al estar probando con gat, utilizamos el niter igual a 50.000 ya que N (600.000) / 50.000 da 12. De esta forma, a cada uno de los cores le correspondrá hacer 3 tareas. El objetivo de que este numero de tareas sea multiplo del numero de cores es que, de no serlo, en la última iteración habrían cores que no tendrían trabajo que hacer y estaríamos desaprovechando tiempo de ejecución ya que estarían parados.

El siguiente bucle sí que es paralelizable, pero al ser un bucle que va hasta fk, que es 200, nos parece que es un número poco significativo y no compensa paralelizar.

El siguiente bucle presenta un conflicto en cuanto al acceso a los vectores fS y fA debido a que el índice es otro vector. Para solucionar esto, propusimos la utilización de la clausula atomic para estos casos. Sin embargo, establecer este código como atomic implica que solo puede ejecutarlo un thread al mismo tiempo, por lo que al final esto conllevaba un incremento en el tiempo de ejecución por lo que no merece la pena.

El último bucle presenta problemas en cuanto a la variable escalar dif. Nuestra propuesta fue la utilización de la clausula reduction con el objetivo de sumar, al final de la ejecución, los valores de la variable dif que se copia a cada uno de los threads de forma privada. Esto supuso un incremento en el speedup, pero el numero de iteraciones que se ejecutaban de esta forma diferia respecto al algoritmo secuencial, pese a que el resto de valores eran correctos. Es por ello que decidimos abandonar la opción de paralelizar este bucle.

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.

dynamic parallel for de i=0 a fN con 50000 iteraciones cada thread

  S1: min = 0

  S2: diff = abs(fV[i] - fR[0])

  S3,S4,S5: realizar segundo bucle en secuencial

  S6: fD[i] = min

end parallel for

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];

    do
     {
      #pragma omp parallel for schedule(dynamic, 50000)
      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;
      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);
}
```

En este apartado únicamente copiamos el código de la función kmean ya que es la única que hemos modificado, el resto del código está igual que el original. La explicación de la paralelización y nuestras decisiones viene explicada en el apartado de propuesta de paralelización.

SpeedUP

Ejecución en gat

Num threads	Elapsed Time (s)	%CPU	SpeedUP
2	28,04	197	2,3235
4	14,43	386	4,5149
8	15,26	363	4,2693
16	15,36	360	4,2415

Ejecución en roquer

Num threads	Elapsed Time (s)	%CPU	SpeedUP
2	10,24	199	2,0781
4	5,34	398	3,9850
8	5,01	676	4,2475
16	5,14	695	4,1401

El máximo SpeedUP conseguido en la máquina gat corresponde a 4 threads, ya que esta tiene 4 cores, siendo este de 4.5149.

En la máquina Roquer se alcanza el máximo SpeedUP con 8 threads. Sin embargo, cabe destacar que el % de uso de la CPU no es cercano al 800%. Esto es debido a lo que explicamos en el apartado de propuesta: decidimos usar un niter = 50.000 porque este es múltiplo de 4, que es el número de cores que tiene la máquina gat ya que ahí es donde queríamos buscar el máximo SpeedUP. Esto da lugar a 12 tareas, número que no es múltiplo de 8, por lo que en la segunda repartición de tareas a los threads, 4 de ellos se encuentran parados sin realizar trabajo. Es por esto que no se alcanza un valor cercano a 800%. Para una implementación más escalable, en lugar de 50.000, se debería fijar niter a la fórmula N/(num_threads*3) o también ppdría ser *2, para que el número de tareas sea multiplo del numero de threads y no perdamos capacidad de ejecución.

Como añadido extra, al probar la paralelización del tercer bucle de la función kmean utilizando la cláusula atomic, nos dio un tiempo de 21.56 segundos. Está dentro del SpeedUp requerido pero, como ya hemos comentado en la propuesta, decidimos descartarlo porque sin él el SpeedUp es mayor.