PRÁCTICA 1.2
Computación Paralela y Masiva

Marcos Esteve Hernández

Geovanny Alexander Risco Carrera

Indice

Enunciado de la práctica

Càlcul d'una regressió lineal amb l'algorisme Gradient-Descent . A partir d'una col·lecció de punts x,y aplicar l'algorisme iteratiu Gradient-Descent aproximant els valors a i b (y=a+bx) fins la seva convergencia.

Temps d'execució seqüencial amb les opcions "-O3" per N=2.000.000:

a Gat -> 78.11 segons,

a Roquer -> 16.35 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.

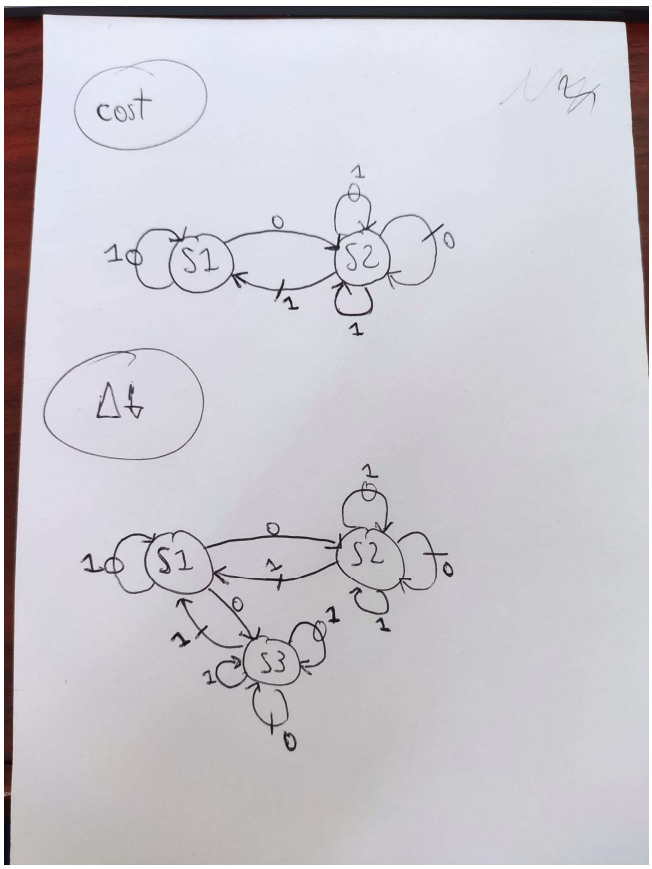
Análisis de dependencia de datos

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

El primero de los bucles que analizamos es el de la función cost:

```
for(i=0;i<nn;i++)
   {
    val = t0 + t1*vx[i] - vy[i];
    sum += val * val;
   }
```

El grafo de dependencias que corresponde a este bucle es el primero de la siguiente imagen:

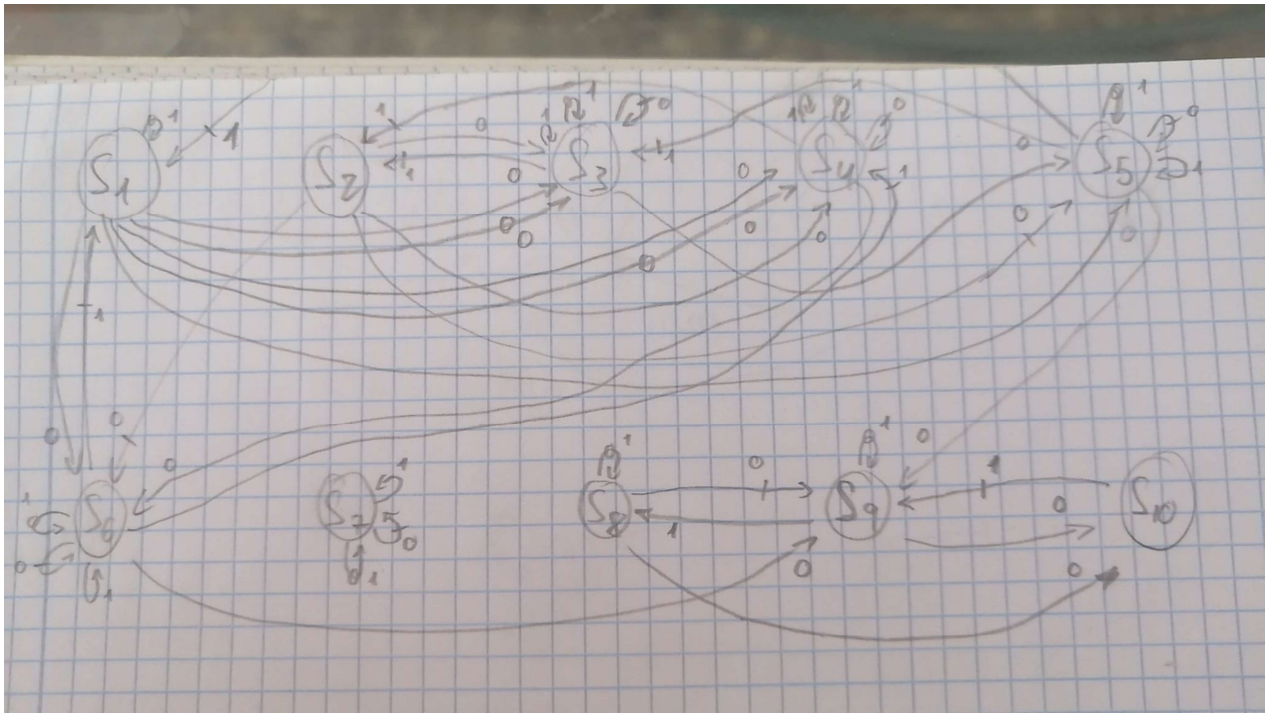


En este grafo observamos que solo hay 2 sentencias, que presentan dependencias tanto consigo mismas como entre ellas, a distancias de 0 y 1. Esto imposibilita la paralelización, aunque en el apartado de propuesta detallaremos cómo la hemos llevado a cabo.

El siguiente código por analizar es el que pertenece a la función gradientDescent:

```
do
   {
    z0 = z1 = 0.0;
    #pragma omp parallel for reduction (+:z0,z1) private (val)
    for(i=0;i<nn;i++)
     {
      val = t0 + t1*vx[i] - vy[i];
      z0 += val;
      z1 += val * vx[i];
     }
    t0 -= z0 * a_n;
    t1 -= z1 * a_n;
    iter++;
    ca = c;
    c = cost(nn,vx,vy,t0,t1);
   }
  while (fabs(c - ca) > error);
```

Primeramente, hemos analizado el bucle for interno, pero también hemos analizado todo el bucle do while, incluyendo el análisis del bucle for en el grafo global:



En este grafo están incluidas las dependencias que se ven en el segundo grafo de la primera imagen, pero también el resto de las dependencias del código. Por ello, únicamente paralelizaremos el bucle for interno, de una manera similar a la paralelización del bucle de la función cost.

Propuesta de paralelización

Como hemos introducido brevemente en el anterior apartado, el primero de las paralelizaciones que realizamos es en el bucle de la función cost. Las dependencias entre las dos sentencias de distancia 0 no nos afectan, ya que el interior del bucle se ejecuta en secuencial. Sin embargo, las dependencias de distancia 1 sí que afectan en la paralelización. Para solucionar estas, lo que realizamos es definir las variables en conflicto como privadas, de forma que se haga una copia en cada thread y así no entren en conflicto entre ellos. Como la variable sum contiene la suma acumulada de los valores calculados en cada iteración, nos interesa conocer el valor final para esta, no los valores que ha calculado cada thread por separado; es por ello por lo que utilizamos la cláusula reduction con la operación de suma, para obtener el valor final de la misma forma que si lo ejecutásemos en secuencial.

Para el segundo bucle que vamos a paralelizar, el bucle for de la función gradientDescent, hemos propuesto una paralelización similar al ya comentado. En este caso, tenemos igualmente dependencias de distancia 0 que no nos afectan ya que el cuerpo del bucle se ejecuta en secuencial. Para las de distancia 1, hemos definido también como privadas las variables que presentan conflicto. De igual forma que ocurre en el anterior bucle con la variable sum, en este las variables z0 y z1 son las que guardan el valor acumulado de las iteraciones, por lo que es a estas a las que hemos aplicado la clausula reduction con la operación suma.

Pseudocódigo de la propuesta

Función “cost”:

static parallel for de i=0 a nn con variables privadas i y val, y reduction(+) de sum

S1: val = t0 + t1*vx[i] - vy[i]

S2: sum = sum + val * val

end parallel for

Función “gradientDescendent”:

static parallel for de i=0 a nn con variables privadas i y val, y reduction(+) de z0 y z1

S1: val = t0 + t1*vx[i] - vy[i]

S2: z0 = z0 + val

S3: z1 = z1 + val * vx[i]

end parallel for

Código + paralelización

```
double cost (int nn, double vx[], double vy[], double t0, double t1)
 {
  int i;
  double val,sum=0.0;

  #pragma omp parallel for reduction (+:sum) private (val)
  for(i=0;i<nn;i++)
   {
    val = t0 + t1*vx[i] - vy[i];
    sum += val * val;
   }
  sum /= 2*nn;
  return(sum);
}

int gradientDescent (int nn, double vx[], double vy[], double alpha, doub
le *the0, double *the1)
 {
  int i;
  double val;
  double z0,z1;
  double c=0,ca;
  double t0=*the0, t1=*the1;
  double a_n = alpha/nn;
  int iter = 0;
  double error = 0.000009; // cinc decimals
  do
   {
    z0 = z1 = 0.0;
    #pragma omp parallel for reduction (+:z0,z1) private (val)
    for(i=0;i<nn;i++)
     {
      val = t0 + t1*vx[i] - vy[i];
      z0 += val;
      z1 += val * vx[i];
     }
    t0 -= z0 * a_n;
    t1 -= z1 * a_n;
    iter++;
    ca = c;
    c = cost(nn,vx,vy,t0,t1);
   }
  while (fabs(c - ca) > error);
  *the0 = t0;
  *the1 = t1;
  return(iter);
}
```

SpeedUP

Ejecución en gat

Num threads	Elapsed Time (s)	%CPU	SpeedUP
2	44,05	197	1,7732
4	22,75	395	3,4334
8	29,13	366	2,6814
16	32,16	366	2,4288

Ejecución en roquer

Num threads	Elapsed Time (s)	%CPU	SpeedUP
2	15,84	199	1,0322
4	15,90	398	1,0283
8	16,05	796	1,0187
16	16,08	655	1,0168

El máximo SpeedUp conseguido en GAT corresponde a la ejecución con 4 threads y es de 3,4334.

En Roquer, por otro lado, no hemos conseguido un SpeedUp para nada significativo, siendo el que mejor el correspondiente a 2 threads. No estamos seguros del motivo de esto, ya que si se aprecia que hay un incremento del uso de la CPU, pero es posible que por motivos de sincronización u overhead no se pueda alcanzar mayor rendimiento.