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 2.2:

Càlcul d'una regressió lineal amb l'algorisme Gradient-Descent.

Estudiants: Leandro Favio Gomez Racca

Miriam Gertrudix Pedrola

Professor: Carles Aliagas Castell

Índex:

Enunciat

El codi és el mateix que la pràctica P1.2 canviant la N de 2.000.000 a 5.000.000

El temps seqüencial a POP és de 140 segons.

- Es programarà amb MPI.
- L'execució es farà per els nombre de processos que indica la taula (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.
- La solució del sistema ha de quedar emmagatzemada al procés 0.
- L'speedup en qualsevol configuració ha de ser superior a 1.75
- L'speedup harmonic a POP ha de ser superior a 3.5

Proposta:

La proposta de paral·lelització que fem està basada en els dos conceptes primordials dels algorismes paral·lels: Balancejar la càrrega entre tots els nodes equitativament i reduir la comunicació entre ells.

Després d'estudiar les diferents seccions del codi, vam decidir:

La nostra idea era aconseguir que els bucles for que havíem paral·lelitzat en la pràctica 1.2 es poguessin fer de forma paral·lela amb la programació MPI sense que això afectes el resultat.

Per a fer-ho vam decidir repartir el for de la funció Cost en seccions iguals perquè cada procés fes una porció d'aquest. El problema ens el generava la variable sum, ja que el programa necessitava la suma de tots els valors d'aquesta variable que generes cada procés. El que vam decidir va ser fer és un reduce a la variable sum. El resultat correcte l'obtenia el procés 0.

Per al bucle for de la funció gradientDescent se'ns complicava una mica més. Volíem utilitzar la mateixa estratègia que en el cas anterior, però sense que això afectes el correcte funcionament del bucle do while.
Per això el que fem és que aquesta vegada en fer el reduce de la variable z0 i z1, el resultat correcte el tinguin tots els processos. D'aquesta manera ens assegurarem que els valors de t0 i t1 en futures iteracions siguin correctes per a tots els processos.

I per a solucionar el problema de la condició del bucle do while, donat que la variable c només és correcta per al procés 0. El que farem és un broadcast de la condició per a tots els processos, i així assegurarem que tots els processos facin les iteracions corresponents.

El print amb els resultats només el farà el procés 0.

Pseudocodi:

```
N 5000000

double X[N];
double Y[N];
enter id_proces;
enter num_procesos;

funcio cost (enter nn, double vx[], double vy[], double t0, double t1) retorna double
  enter i;
  double val,sum := 0.0, sum_global;

  per (i := (id_proces) * (nn/num_procesos) ; i < (nn/num_procesos) * (id_proces + 1) ; i
:= i+1)
  {
    val := t0 + t1*vx[i] - vy[i];
    sum := sum +(val * val);
  }
MPI_Reduce( sum, sum_global);
  sum_global:= sum_global/(2*nn);
  retorna (sum_global);
 ffuncio

funcio gradientDescent (enter nn, double vx[], double vy[], double alpha, double *the0,
double *the1) retorna enter

  enter i;
  double val;
  double z0,z1,z0_global, z1_global;
  double c := 0,ca;
  double t0 := *the0, t1 := *the1;
  double a_n := alpha/nn;
  enter iter := 0;
  double error := 0.000009; // cinc decimals

  fer
 z0 := z1 := 0.0;
 per (i := (id_proces) * (nn/num_procesos) ; i < (nn/num_procesos) * (id_proces + 1) ; i
:= i+1) fer
      val := t0 + t1*vx[i] - vy[i];
      z0 := z0 + val;
      z1 := z1 + (val * vx[i]);
    fper
MPI_Allreduce( z0, z0_global);
MPI_Allreduce( z1, z1_global);
```

```
    t0 := t0 - (z0_global * a_n);
    t1 := t1 - (z1_global * a_n);
    iter := iter + 1;
    ca := c;
    c := cost(nn,vx,vy,t0,t1);
    condition = fabs(c - ca) > error;
    Broadcast (condition)
  mentre (condition );
  *the0 := t0;
  *the1 := t1;
  retorna (iter);
ffuncio


funcio principal()

enter i;
double ct;
double theta0=0, theta1=1;

  srand(1);
  per (i := 0 ; i<N ; i := i+1) fer
    X[i] := frand(13);
    Y[i] := frand(9) + ((1.66 + (frand(0.9))) *  X[i]) * X[i] ;
  fper

  //for (i=0;i<N;i++) printf("%g %g\n",X[i],Y[i]);

  i := gradientDescent (N, X, Y, 0.01, &theta0, &theta1);
  ct := cost(N,X,Y,theta0,theta1);
si (id_proces == 0) llavors
  printf (i "Theta;" theta0 "," theta1  "cost:" ct);
fsi
ffuncio
```

Codi:

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


#define frand(M) (M*(((double)rand())/RAND_MAX))


#define N 5000000


double X[N];
double Y[N];
/* rank del proces    */
int    el_meu_rank;
/* numero de processos        */
int    world_size;


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


 for(i=(el_meu_rank)*(nn/world_size);i<(nn/world_size)*(el_meu_rank+1);i++)
 {
   val = t0 + t1*vx[i] - vy[i];
   sum += val * val;
 }
 MPI_Reduce(&sum, &sum_global, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
 sum_global /= 2*nn;
 return(sum_global);
}
```

```
int gradientDescent (int nn, double vx[], double vy[], double alpha, double
*the0, double *the1)
{
 int i;
 double val;
 double z0, z1, z0_global, z1_global;
 double c=0,ca;
 double t0=*the0, t1=*the1;
 double a_n = alpha/nn;
 int iter = 0, condition;
 double error = 0.000009; // cinc decimals

 do
 {
   z0 = z1 = 0.0;

for(i=(el_meu_rank)*(nn/world_size);i<(nn/world_size)*(el_meu_rank+1);i++)
    {
     val = t0 + t1*vx[i] - vy[i];
     z0 += val;
     z1 += val * vx[i];
    }
   MPI_Allreduce(&z0, &z0_global, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   MPI_Allreduce(&z1, &z1_global, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   t0 -= z0_global * a_n;
   t1 -= z1_global * a_n;
   iter++;
   ca = c;
   c = cost(nn,vx,vy,t0,t1);
   condition = fabs(c - ca) > error;
   MPI_Bcast(&condition,1,MPI_INT,0,MPI_COMM_WORLD);
 }
 while (condition);
 *the0 = t0;
```

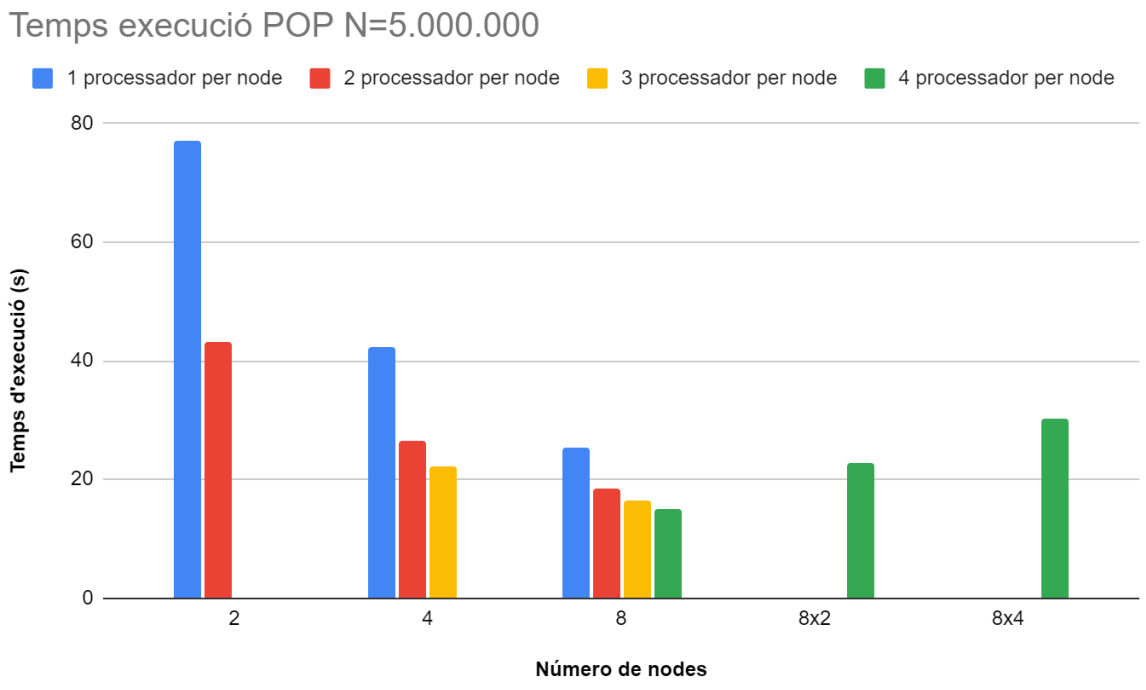
```
 *the1 = t1;
 return(iter);
}

int main()
{
 /* Inicialitzar MPI */
 MPI_Init(NULL, NULL);
 /* 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, &world_size);

 int i;
 double ct;
 double theta0=0, theta1=1;

 srand(1);
 for (i=0;i<N;i++)
  {
   X[i] = frand(13);
   Y[i] = frand(9) + ((1.66 + (frand(0.9))) *  X[i]) * X[i] ;
  }

 //for (i=0;i<N;i++) printf("%g %g\n",X[i],Y[i]);

 i=gradientDescent (N, X, Y, 0.01, &theta0, &theta1);
 ct=cost(N,X,Y,theta0,theta1);
 if (el_meu_rank == 0)
 {
   printf ("(%d) Theta; %g, %g  cost: %g\n",i,theta0,theta1,ct);
 }
```

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

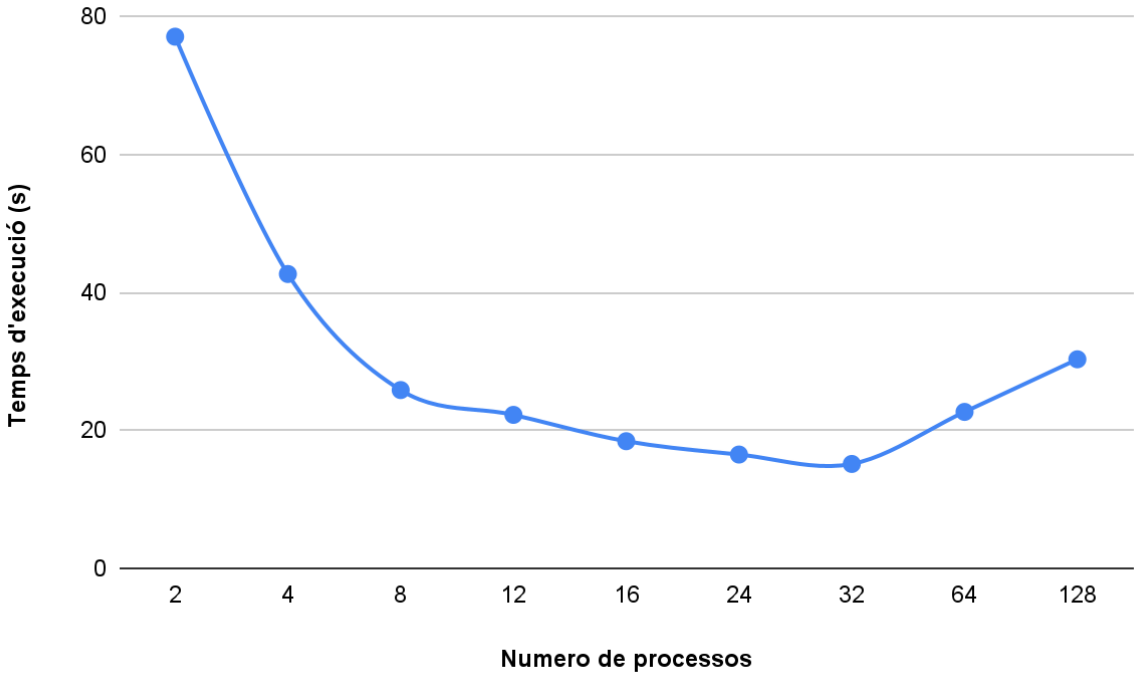
Resultats i conclusions:

A continuació analitzem l'Speed up del nostre codi paral·lelitzat per a N=5.000.000
El Speed up harmònic a POP ens ha donat un valor de **4,54**. Aquest Speed Up l'hem calculat fent la mitjana harmònica de tots els Speed ups obtinguts.
La mitjana aritmètica de tots els temps d'execució ens ha donat: **30,84s**

En aquest gràfic veiem en color blau el temps d'execució per 1 processador per node, en color vermell 2 processos per node, en groc 3 processos per node i en verd 4 processos per node. En l'eix horitzontal tenim el nombre de nodes.



D'aquesta gràfica n'hem tret les següents conclusions. Com ja esperàvem a causa de la programació aplicada al nostre codi, l'augment de quantitat de processos fa que disminueixi el temps (es pot veure la tendència decreixent de la gràfica). Però si ho observem amb més atenció podem veure que aquesta disminució de temps no és lineal. Hem decidit elaborar una gràfica d'aquest comportament per explicar-ho millor.



Aquí podem veure més clarament aquesta disminució del temps. Com veiem al principi la diferencia en afegir processos és molt elevada, és a dir, és molt més ràpid executar el codi amb dos processos que en seqüencial, així com ho és amb 4 processos en comptes de 2. Però a mesura que tenim una quantitat molt gran de processos aquesta diferència de temps cada vegada és més petita. Això és degut al fet que la major part del temps d'execució s'està invertint en la part seqüencial del codi, això vol dir que arribat a cert punt afegir processos no implica una millora. Arribant a un punt en què fer tota la comunicació entre processos fa que el temps d'execució augmenti. Com és el cas de 8x2 i 8x4 nodes (64 i 128 processos).

Com a coses que hem observat i millores que podríem aplicar al codi, hem vist que falla en el moment que intentem posar un nombre imparell de processos, a causa de la repartició dels bucles. I una cosa que podríem millorar és buscar la manera que la part seqüencial la fes un sol procés en comptes de tots, això no afectaria el rendiment, però sí que reduiria el cost computacional i també el cost elèctric.