Problema 1 - Prelucrarea unei imagini blurate

12 aprilie 2022

1 Introducere

Scopul acestei probleme este scrierea unui program MPI care să implementeze un algoritm simplu de procesare a imaginilor (reconstituirea unei imagini blurate cu un algoritm de detecție a limitelor de zonă de culoare).
Veți începe prin a scrie codul serial, astfel încât paralelizarea să fie simplă. Formatul grafic PGM este în esență o matrice - abordarea paralelă naturală va fi de tip descompunere pe domenii. Comunicarea MPI va implica doar procesele vecini de ordin I pe grid și va conține apeluri ale funcțiilor **MPI_Scatter()** și **MPI_Gather()**.
Veți descompune problema în "felii" (adică veți utiliza un grid de procese 1D sau altfel spus o topologie de procese inelară, împărțind numărul de linii de pixeli din fișierul PGM intre procese), a se vedea figura 1.

Fișierul PGM este în esență un fișier text (ASCII) cu un antet de tipul:

P2
XSIZE YSIZE
MAXGREY

Urmează toate cele $XSIZE * YSIZE$ valori dintr-o matrice de întregi, separate de spații. Lungimea unei linii nu trebuie să depașească 70 de caractere, inclusiv spațiile.

$P2$ de pe prima linie este "numărul magic" care identifică formatul fișierului (aici, PGM plain; a se vedea pagina de manual "man pgm" într-un terminal Linux). Valorile $XSIZE$ (numărul de pixeli pe orizontală) și $YSIZE$ (numărul de pixeli pe verticală) indică dimensiunea matricii, iar $MAXGREY$ indică valoarea maximă a contrastului – 0 înseamnă culoarea neagră, $MAXGREY$ înseamnă culoarea albă, valori intermediare indică nuanțe de gri.

2 Algoritmul

Fișierul **image_640x480.pgm** reprezintă rezultatul unui algoritm simplu de detecție a marginilor unor zone dintr-o imagine (edge-detection algorithm). Algoritmul a fost aplicat unei imagini în nuanțe de gri, de dimensiune $M \times N$ ($M = 640,\ N = 480$). Pixel-ul de margine este definit ca:

$$plim[i][j] = img[i-1][j]+img[i+1][j]+img[i][j-1]+img[i][j+1]-4*img[i][j].$$

Dacă un pixel are aceeași valoare ca și cei 4 pixeli vecini (deci nu este o margine de zonă), valoarea $plim[i][j]$ va fi zero. Daca pixel-ul analizat este foarte diferit de cei patru vecini, (deci aparține unei posibile margini de zonă), $plim[i][j]$ va fi mare în valoare absolută. Valorile i și j sunt în intervalul $1, 2, \ldots, M$ și respectiv $1, 2, \ldots, N$. Pixelii din afara acestei regiuni (de ex. $img[i][0]$ sau $img[M+1][j]$, care aparțin regiunilor de halo, a se vedea mai jos) sunt considerați a fi albi, deci au valoarea 255.

Problema propusă face de fapt operația inversă: reconstruiește imaginea pe baza fișierului care conține structura de margini de zonă.
Justificarea paralelizării e dată de numărul mare de operații de efectuat, inclusiv schimbul de date între procese. În plus, tehnica transferului de date între procese care rulează în paralel via regiuni de halo (regiuni în care un proces preia date de la un alt proces în vederea îndeplinirii sarcinilor; în cazul de față fiecare proces are nevoie de liniile/coloanele marginale de pixeli de la procesele cu ranguri vecine pentru a aplica algoritmul de reconstrucție).

3 Codul serial

Imaginea poate fi reconstruită prin operații repetate de forma:

$$pnew[i][j] = 0.25*(pold[i-1][j]+pold[i+1][j]+pold[i][j-1]+pold[i][j+1]-plim[i][j])$$

in care *pold* și *pnew* sunt valorile pixelilor la începutul și sfârșitul unei iterații. Valorile inițiale ale matricilor de pixeli $pold[M+2][N+2]$, $pnew[M+2][N+2]$, $plim[M+2][N+2]$ vor corespunde culorii albe (adica 255, important pentru definirea culorii din regiunea de halo, schimbată între procese în codul paralel). Regiunea de halo conține două linii și două coloane suplimentare față de buffer-ul $data[M][N]$ în care se stochează imaginea inițială (fără halo).

Structura codului:

$float\ pold[M+2][N+2]$, $pnew[M+2][N+2]$, $plim[M+2][N+2]$

$float\ data[M][N]$

1. extrage valorile M, N din fișierul grafic (format pgm), folosind funcția $pgm\_size()$
2. citește fișierul conținând imaginea marginilor de zonă în matricea $data[M][N]$, folosind funcția pgm_read()
3. pentru $i = 1, M$; $j = 1, N$
   $plim[i][j] = data[i-1][j-1]$
4. atribuie 255 tuturor elementelor matricilor *pold*, *pnew*, *plim*, inclusiv halo-ului
5. începe iterarea (numărul de iterații *niter* poate fi specificat în linia de comandă, ca și numele fișierului grafic inițial, conținând marginile de zonă sau pot fi specificate în cod):
   pentru $i = 1, M$; $j = 1, N$
   $pnew[i][j] = 0.25*(pold[i-1][j]+pold[i+1][j]+pold[i][j-1]+pold[i][j+1] - plim[i][j])$
   copiază matricea *pnew* în *pold*, **fără copierea valorilor de halo**
6. termină iterarea
7. copiază matricea *pold* în *data*, fără valorile de halo
8. scrie fișierul PGM apelând pgm_write()

Funcțiile auxiliare $pgm\_size()$, $pgm\_read()$, $pgm\_write()$ sunt date în fișierul $pgm\_IO.c$, care conține la început o descriere a modului de utilizare a acestor funcții; în principiu, e suficient să citiți această parte, dar puteți examina codul în întregime. Veți include în codul dumneavoastră:
$\#include$ "$pgm\_IO.h$"
$\#include$ "$pgm\_IO.c$"

Testați-vă programul și produceți fișierele imagine reconstruite cu 20, 200, 400, 600, 800, 1000, 1200 și 1400 iterații. Puteți vizualiza rapid rezultatul, pe un sistem Linux, folosind aplicația *display*, sub forma $display\ fisier.pgm$; a se vedea și pagina de manual *man display*.

4 Codul paralel

Paralelizați codul serial (dupa ce îl testați!), divizând imaginea între procese de-a lungul liniilor de pixeli: $MP = M/nproc$, $NP = N$ (Fig. 1).

Comunicația MPI implică transmiterea și recepția unor linii întregi de date (e comod de utilizat **MPI_Sendrecv()**):

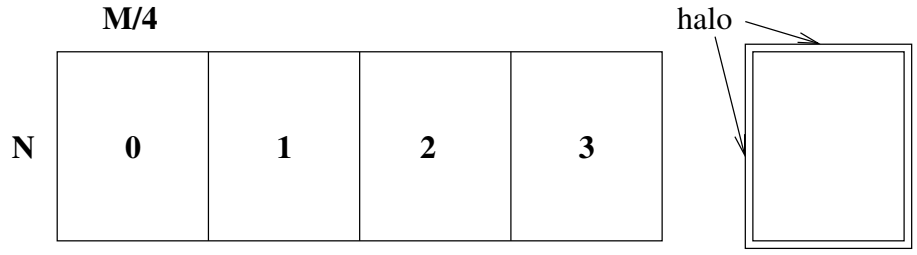

Figura 1: Distribuirea între procese rulând în paralel a liniilor de pixeli ai matricii grafice (stânga) și structura matricii (blocului) alocat unui proces, incluzând regiunea de halo (dreapta). Aici se presupune că au fost lansate în execuție 4 procese.

- procesul master (rangul 0) creează o matrice $masterdata[M][N]$ în care preia cu $pgm\_read()$ conținutul fișierului grafic inițial
- toate procesele creează matricile $data[MP][NP]$, $pold[MP+2][NP+2]$, $pnew[MP+2][NP+2]$, $plim[MP+2][NP+2]$
- distribuie segmentele de date $MP * NP$ din $masterdata$ către toate procesele, folosind $MPI\_Scatter()$
- inițializează toate matricile locale și iterează algoritmul de reconstrucție (la fel ca în codul serial), incluzând în bucla de iterare:
  - trimite cele NP elemente ($pold[MP][j]$; $j = 1, NP$) către procesul cu rangul $rank + 1$
  - recepționează NP elemente de la procesul cu rangul $rank - 1$ în ($pold[0][j]$; $j = 1, NP$)
  - trimite NP elemente ($pold[1][j]$; $j = 1, NP$) către $rank - 1$
  - recepționează NP elemente de la $rank + 1$ în ($pold[MP + 1][j]$; $j = 1, NP$)
  - aplică algoritmul de reconstrucție
- copiază în $data$ rezultatul, pe fiecare proces
- adună toate matricile locale $data$ în matricea $masterdata$ la procesul master, folosind $MPI\_Gather()$
- procesul master scrie datele din $masterdata$ în fișierul grafic (al cărui nume va conține numărul de iterații efectuat), folosind $pgm\_write()$.
- termină mediul MPI

Lista funcțiilor MPI pe care le-ați putea utiliza este:

- **MPI_Init()**
- **MPI_Comm_size()**
- **MPI_Comm_rank()**
- **MPI_Scatter()**, pentru distribuția datelor de la procesul master (procesul de rang 0), care citește fișierul grafic
- **MPI_Sendrecv()**, pentru transmiterea pixelilor din coloanele de margine către regiunile de halo ale proceselor vecine din inel, cum este explicat mai sus; inelul este deschis: pentru procesele de margine (procesele de rang 0 și $nproc - 1$) puteți utiliza ca sursă/destinație în $MPI\_Sendrecv()$ valoarea de rang specială MPI_PROC_NULL (echivalentă cu /dev/null în Linux/Unix). Utilizarea ei pentru procese cu valori de rang de margine evită cod suplimentar pentru a ne asigura că procesele de margine nu încearcă să trimită/recepționeze date spre/de la ranguri nevalabile (de ex. $my\_rank = -1$ sau $my\_rank = nproc$)
- **MPI_Gather()**, pentru a colecta la procesul master datele procesate de procesele slave
- **MPI_Finalize()**

Succes!