ALGORITMI PARALELI ȘI DISTRIBUIȚI
Tema #1 Generare paralelă de fractali folosind mulțimile Mandelbrot și Julia

Termen de predare: 15-11-2020 23:59

Cerință

Pornind de la implementarea secvențială, se cere să se scrie un program paralel care calculează mulțimile Mandelbrot[1] și Julia[2] pentru o funcție polinomială complexă de forma $f(z) = z^2 + c$ şi le afişează sub formă de imagini grayscale. Programul va fi scris în C/C++ și va fi paralelizat utilizând PThreads.

Definiții

Mulțimea Mandelbrot

Fie familia de polinoame complexe $P_c : \mathbb{C} \to \mathbb{C}$, definite de $P_c(z) = z^2 + c$, cu c un număr complex. Se definește mulțimea Mandelbrot ca fiind totalitatea punctelor c pentru care secvența $0, Pc(0), Pc(Pc(0)), ...$ nu tinde către infinit:

$$M = \{c \mid \exists\ s \in \mathbb{R}\ a.i.\ \forall\ n \in \mathbb{N},\ |P_c^n(0)| < s\} \tag{1}$$

Generarea și reprezentarea mulțimii Mandelbrot se poate realiza folosind următorul algoritm:

```
foreach c in the complex plane do
   z = 0 + 0i
   step = 0
   while |z| < 2 and step < MAX_STEPS do
       z = z * z + c
       step = step + 1
   color = step mod NUM_COLORS
   plot(c.x, c.y, color)
```

Mulțimea Julia

Fie $f(z) : \mathbb{C} \to \mathbb{C}$, $f(z) = \frac{P(z)}{Q(z)}$ o funcție rațională complexă. Mulțimea Julia plină J_f a funcției este mulțimea punctelor din planul complex care au o orbită mărginită în raport cu f (unde $f^n(z)$ este $f(f(f(...(z))))$ de n ori):

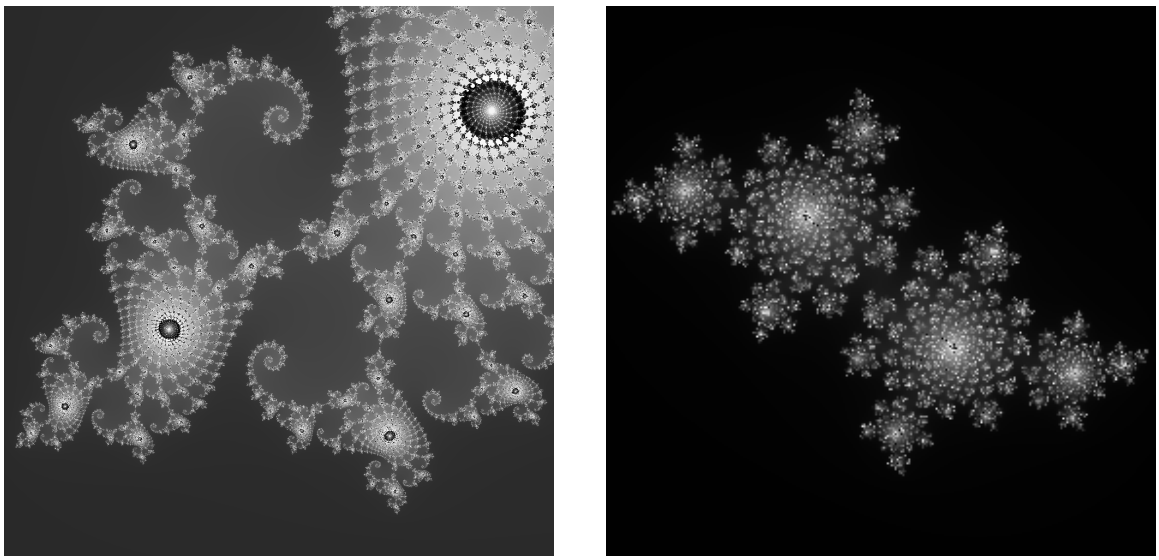
$$J_f = \{z \in \mathbb{C} \mid \exists\ s \in \mathbb{R}\ a.i.\ \forall\ n \in \mathbb{N},\ |f^n(z)| < s\} \tag{2}$$

[1] https://mathworld.wolfram.com/MandelbrotSet.html
[2] https://mathworld.wolfram.com/JuliaSet.html

Generarea și reprezentarea mulțimii Julia pline pentru o funcție $f(z) = z^2 + c$ se poate realiza folosind următorul algoritm:

```
foreach z in the complex plane do
   step = 0
   while |z| < 2 and step < MAX_STEPS do
       z = z * z + c
       step = step + 1
   color = step mod NUM_COLORS
   plot(z.x, z.y, color)
```

În figura de mai jos, se pot observa reprezentările a două mulțimi Mandelbrot și Julia în format de imagini grayscale.



(a) Mandelbrot (b) Julia

Implementarea secvențială

În resursele temei, găsiți un fișier numit **tema1.c** care conține implementarea secvențială a calculului celor două mulțimi și generarea de imagini grayscale. Programul primește următorii parametri la rulare:

```
./tema1 <fisier_intrare_julia> <fisier_iesire_julia> <fisier_intrare_mandelbrot>
    <fisier_iesire_mandelbrot>
```

Se poate deci observa că o rulare a programului secvențial va calcula ambele mulțimi pe baza unor fișiere de intrare specifice fiecăreia. Pentru Mandelbrot, un exemplu de fișier de intrare are următorul format:

```
0
-2.5 1.0 -1.0 1.0
0.001
5000
```

Prima linie specifică tipul mulțimii, adică 0 pentru Mandelbrot. A doua linie conține limitele axelor OX și OY pe care se realizează calculele (mai exact, subspațiul complex în care se lucrează). Cea de-a treia linie specifică rezoluția de calcul, adică pasul cu care se iterează prin planul complex. Ultima linie reprezintă numărul maxim de pași de calcul, adică variabila MAX_STEPS din pseudocodul de mai sus. Astfel, fișierul de intrare dat ca exemplu va genera mulțimea Mandelbrot între -2.5 și 1.0 pe axa OX, respectiv între -1.0 și 1.0 pe axa OY, cu o rezoluție de 0.001. În cadrul algoritmului, se va folosi un număr de maximum 5000 iterații.

Un exemplu de fișier de intrare pentru Julia este prezentat mai jos:

```
1
-2.0 2.0 -2.0 2.0
0.001
5000
-0.6 0
```

Așa cum se poate observa, există două diferențe față de Mandelbrot. Mai exact, prima linie conține 1, care specifică faptul că se va genera o mulțime Julia, pentru care se mai adaugă o linie adițională la final care conține părțile reale și imaginare ale parametrului c al funcției. Astfel, fișierul de intrare de mai sus va genera mulțimea Julia a funcției f între -2.0 și 2.0 pe axa OX, respectiv între -2.0 și 2.0 pe axa OY, cu o rezoluție de 0.001. În cadrul algoritmului, se va folosi un număr de maximum 5000 iterații.

Pașii de funcționare ai implementării secvențiale pe care o găsiți în resursele temei sunt următorii:

1. se citesc argumentele programului
2. se citește fișierul de intrare specific calculului mulțimii Julia
3. se alocă memorie pentru rezultatul calculului mulțimii Julia
4. se rulează algoritmul care calculează mulțimea Julia și se scrie rezultatul într-o matrice
5. se scrie rezultatul calculului mulțimii Julia într-un fișier de tip PGM
6. se eliberează memoria alocată pentru calculul mulțimii Julia
7. se citește fișierul de intrare specific calculului mulțimii Mandelbrot
8. se alocă memorie pentru rezultatul calculului mulțimii Mandelbrot
9. se rulează algoritmul care calculează mulțimea Mandelbrot și se scrie rezultatul într-o matrice
10. se scrie rezultatul calculului mulțimii Mandelbrot într-un fișier de tip PGM
11. se eliberează memoria alocată pentru calculul mulțimii Mandelbrot.

Rezultatele programului vor fi scrise în fișiere de ieșire în format PGM "plain"[3]. Imaginile rezultate vor avea un număr de 256 de nuanțe de gri (valori între 0 și 255). Dimensiunile imaginilor finale se calculează pe baza dimensiunilor subspațiului complex în care se lucrează (x_{min}, x_{max}, y_{min} și y_{max}, adică valorile citite de pe a doua linie din fișierul de intrare) și a rezoluției (valoarea citită de pe a treia linie din fișierul de intrare), conform formulelor de mai jos (unde $\lfloor x \rfloor$ reprezintă partea întreagă a lui x, deci valorile de mai jos vor fi numere întregi):

$$latime = \lfloor \frac{x_{max} - x_{min}}{rezolutie} \rfloor \tag{3}$$

$$inaltime = \lfloor \frac{y_{max} - y_{min}}{rezolutie} \rfloor \tag{4}$$

O imagine de tip PGM "plain" are următorul format:

[3] http://netpbm.sourceforge.net/doc/pgm.html

- prima linie: numărul magic specific formatului ("P2")
- a doua linie: lățimea și înălțimea imaginii în pixeli (separate de spațiu)
- a treia linie: valoarea maximă de gri (în cazul de față va fi 255, adică $NUM\_COLORS-1$ din algoritmii prezentați în a doua secțiune)
- următoarele linii: valorile de gri ale pixelilor de pe fiecare linie din imagine, separate prin spații albe.

Pentru a deschide imagini PGM, puteți folosi diverse editoare de imagini (de exemplu, Gimp). Deoarece, în cazul coordonatelor matematice, punctul (0,0) se află în partea de stânga-jos a axelor și axa OY este îndreptată în sus, iar pentru coordonatele ecran, punctul (0,0) se află în partea de stânga-sus și axa OY este îndreptată în jos, înainte de a se salva datele în fișierul de ieșire, trebuie așezate coordonatele Y în ordine inversă. Acest lucru se realizează în implementarea secvențială după calculul fiecărei mulțimi, înainte de scrierea rezultatelor în fișiere.

Paralelizarea calculelor

Cerința principală a temei este paralelizarea calculelor celor două mulțimi, pornind de la implementarea secvențială. Va rezulta astfel un binar numit **tema1_par** care are același comportament ca cel secvențial și care va fi rulat în felul următor (unde P este numărul de thread-uri):

```
./tema1_par <fisier_intrare_julia> <fisier_iesire_julia> <fisier_intrare_mandelbrot>
    <fisier_iesire_mandelbrot> <P>
```

Programul vostru trebuie să paralelizeze atât **calculul mulțimilor Julia și Mandelbrot**, cât și **transformarea rezultatului din coordonate matematice în coordonate ecran**. În implementare, nu se recomandă crearea și join-ul de thread-uri de mai multe ori (temele care pornesc și opresc thread-urile de mai multe ori vor primi o **depunctare de până la 20% din nota finală)**. Astfel, **veți crea și porni cele P thread-uri la început, și le veți folosi pentru toate operațiile paralele**, realizând sincronizare folosind primitivele învățate la laborator. Toate cele P thread-uri vor contribui la calculul mulțimilor și la transformarea coordonatelor.

În implementarea paralelă, trebuie să păstrați următoarea ordine a operațiilor:

1. calculul mulțimii Julia
2. scrierea rezultatului calculului mulțimii Julia în fișierul corespuzător de ieșire
3. calculul mulțimii Mandelbrot
4. scrierea rezultatului calculului mulțimii Mandelbrot în fișierul corespunzător de ieșire.

Pentru testarea corectitudinii implementării, vă recomandăm să folosiți implementarea secvențială ca etalon. Astfel, puteți rula programul secvențial pe un set de fișiere, iar apoi să rulați programul paralel, cu valori diferite pentru P, pe același set de fișiere. Puteți apoi să folosiți utilitarul *diff* pentru a verifica dacă rezultatul rulării paralele este același cu cel al rulării secvențiale.

Notare

Detali despre submiterea temei se vor da în săptămâna 2-6 noiembrie. Tema se va trimite într-o arhivă Zip care, pe lângă fișierele sursă, va trebui să conțină următoarele două fișiere **în rădăcina arhivei**:

- *Makefile* - cu directiva *build* care compilează tema voastră și generează un executabil numit **tema1_par** aflat în rădăcina arhivei

- README - fișier text în care să se descrie pe scurt implementarea temei.

Punctajul este divizat după cum urmează:

- **50p** - scalabilitatea soluției
- **30p** - corectitudinea implementării[4]
- **20p** - claritatea codului și a explicațiilor din README.

Testare

Pentru a vă putea testa tema, găsiți în arhiva temei un set de fișiere de intrare de test (patru pentru Julia și patru pentru Mandelbrot) un script Bash numit **test.sh**, pe care îl puteți rula pentru a vă calcula punctajul. Acest script va fi folosit și pentru testarea automată, singura diferență fiind că atunci se vor rula niște teste în plus pe fișiere de intrare și valori ale lui P adiționale. Pentru a putea rula scriptul așa cum este, trebuie să aveți următoatoarea structură de fișiere:

```
$ tree
.
+-- skel
|   +-- Makefile
|   +-- tema1.c
+-- sol
    +-- Makefile
    +-- tema1_par.c
    +-- test.sh
    +-- tests
        +-- julia1.in
        +-- julia2.in
        +-- julia3.in
        +-- julia4.in
        +-- mandelbrot1.in
        +-- mandelbrot2.in
        +-- mandelbrot3.in
        +-- mandelbrot4.in
```

La rulare, scriptul execută următorii pași:

1. compilează și rulează implementarea secvențială pe cele patru seturi de fișiere de intrare
2. compilează și rulează implementarea paralelă pe cele patru seturi de fișiere de intrare pentru 2, 3 și 4 thread-uri
3. se compară fișierele PGM rezultate în urma rulărilor paralele și cele secvențiale
4. se calculează accelerația pentru primul set de fișiere de intrare (de la varianta secvențială la 2, respectiv 4 thread-uri, și de la 2 la 4 thread-uri)
5. se calculează punctajul final din cele 80 de puncte alocate testelor automate (20 de puncte fiind rezervate pentru claritatea codului și a explicațiilor, așa cum se specifică mai sus).

Atenție! Dacă aveți un calculator cu două core-uri (sau patru cu hyper-threading), va trebui să modificați măsurarea accelerației să nu ia în considerare și testul de 4 thread-uri, pentru că implementarea paralelă nu va scala. Dacă programul nu trece nici măcar unul din testele de scalabilitate, punctajul final va fi 0.

[4] Acest punctaj este condiționat de scalabilitate. O soluție secvențială, deși funcționează corect și dă rezultate bune, nu se va puncta.

Rulare pe cluster

Dacă doriți să vă testați scalabilitatea implementării pe un număr mai mare de core-uri și sunteți limitați de resursele hardware, vă recomandăm să încercați să rulați pe cluster-ul facultății. Pentru a realiza acest lucru, puteți urmări pașii din această secțiune.

Conectare la FEP

Pentru a putea rula aplicații pe cluster, primul pas necesită conectarea la FEP (front-end processor), care reprezintă punctul de acces prin intermediul căruia putem da comenzi în cluster. Pentru a ne conecta, trebuie să dăm următoarea comandă (unde *username* reprezintă numele de utilizator cu care vă logați pe Moodle):

```
$ ssh <username>@fep.grid.pub.ro
```

Dacă avem nevoie să copiem fișiere pe FEP, putem executa următoarea comandă pe mașina noastră locală:

```
$ scp <fisier> <username>@fep.grid.pub.ro:.
```

Pentru conectare, se folosește parola de pe Moodle.

Scriptul de rulare

Varianta cea mai facilă de a rula un job pe cluster este de a-l defini prin intermediul unui script Bash. Un exemplu de script care compilează și rulează această temă se poate observa mai jos:

```
#!/bin/bash
module load compilers/gnu-5.4.0
make
./tema1_par julia.in julia.pgm mandelbrot.in mandelbrot.pgm 4
make clean
```

Se poate observa că prima comandă din script încarcă modulul *compilers/gnu-5.4.0*, pentru a se putea compila tema. Dacă doriți să vedeți modulele disponibile pe cluster, puteți da următoarea comandă:

```
$ module avail
```

Pornirea și analiza unui job

Mașinile din cluster sunt grupate în cozi, care sunt mulțimi de calculatoare cu arhitecturi similare. Pentru a putea vedea cozile din cluster și disponibilitatea lor, se poate executa următoarea comandă:

```
$ qstat -g c
```

Pentru trimiterea unui job spre execuție pe o coadă din cluster, se poate folosi următoarea comandă:

```
$ qsub -cwd -q ibm-nehalem.q script.sh
```

Dacă doriți să vă testați această temă în cluster, vă recomandăm să folosiți coada *ibm-nehalem.q*, care conține mașini cu 24 de nuclee. Putem urmări evoluția unui job folosind comanda de mai jos, care ne arată starea sa (*qw* dacă este în așteptare, *r* dacă este în execuție, iar dacă a terminat de rulat nu va mai apărea în listă):

```
$ qstat
```

Atunci când un job se termină de executat, se vor crea în directorul curent de pe FEP două fișiere, de forma *script.sh.e.ID_JOB* și *script.sh.o.ID_JOB*, care vor conține ceea ce s-a scris la *stderr* și respectiv *stdout*.