Laborator 5 – PPD

Analiza cerintelor

Se considera n polinoame reprezentate prin lista de monoame (reprezentare: lista inlantuita ordonata dupa exponentii monoamelor). Se cere adunarea polinoamelor folosind o implementare multithreading (p threaduri).

Polinoamele se citesc din fisiere – cate un fisier pentru fiecare polinom. Un fisier contine informatii de tip (coeficient, exponent) pentru fiecare monom al unui polinom.

Rezultatul obtinut se scrie intr-un fisier rezultat.

Preconditie: Fisierele nu contin monoame cu coeficient egal cu 0.

Constrangere: Sincronizare la nivel de lista

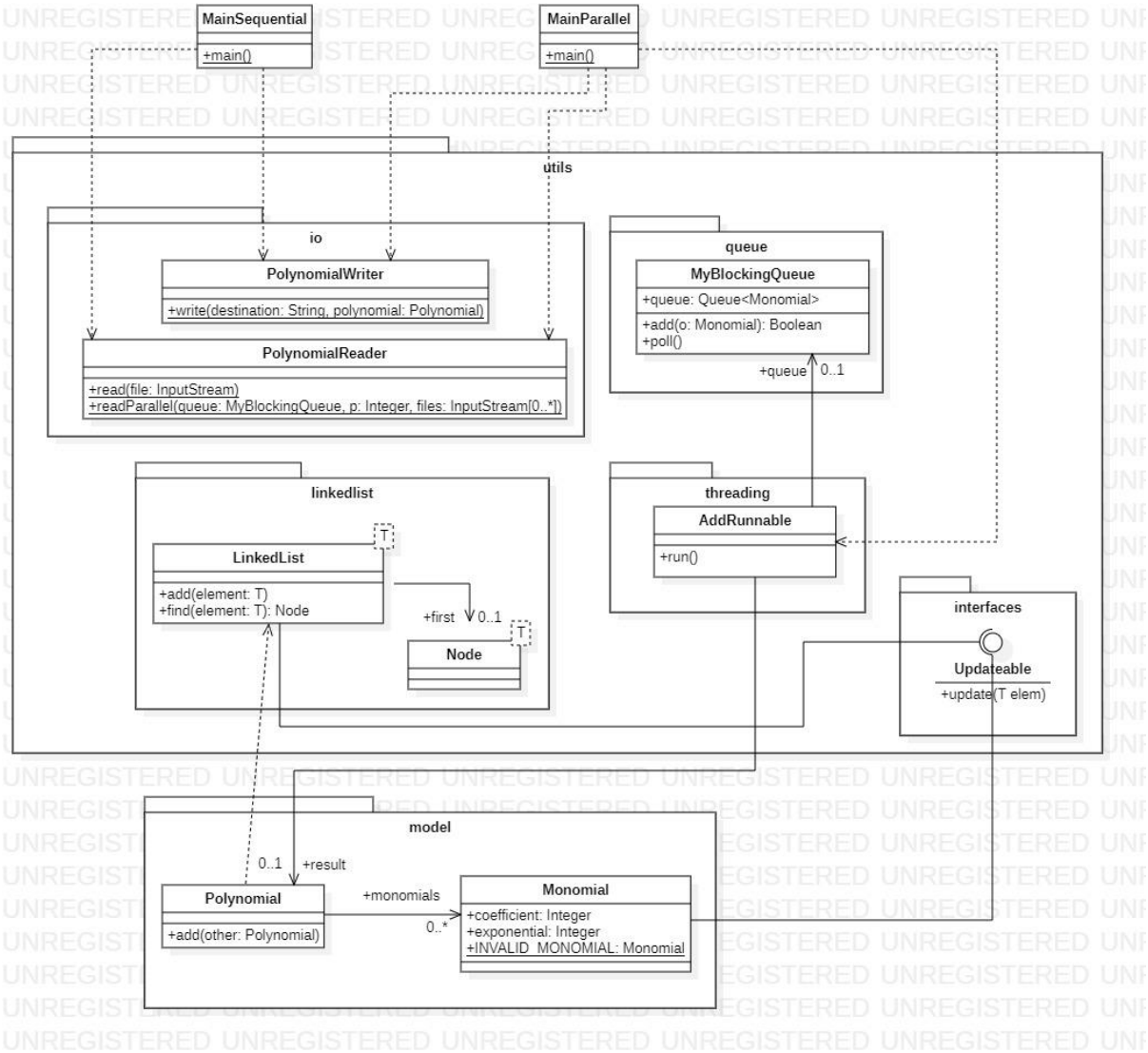
Postconditie: Fisierul rezultat nu contine monoame cu coeficient egal cu 0

Cerinte nefunctionale:

- Folosirea a “p1” thread-uri care vor citi fisierele si a “p2” thread-uri care vor face adunarile efective
- Sincronizarea operatiilor listei inlantuite se vor face la nivel de nod (nu la nivel de intreaga lista) prin mecanisme de tip Lock
- Daca in urma operatiilor rezulta un nod cu coeficient nul, acesta este sters din lista

Proiectare

Diagrama ce prezinta clasele folosite in implementare, impreuna cu metodele lor si relatiile dintre ele.



Detalii de implementare

Tema de laborator a fost realizata in limbajul de programare Java, reutilizand o buna parte din codul care a fost scris pentru Laboratorul 4. Toti parametrii (numar thread-uri, caile catre fisierele cu polinoame) sunt date ca si argumente in linia de comanda.

Clasele din pachetul `model` reprezinta o abstractizare a conceptului de monom si de polinom.

Un polinom contine mai multe monoame, memorate intr-un `LinkedList` din pachetul `utils.linkedlist`, ce reprezinta o implementare proprie a unei liste simplu inlantuie ordonata crescator. Monoamele sunt inserate in aceasta

lista crescator dupa exponentul lor. Lista este sincronizata la nivel de noduri; in metodele de `add`, `delete` si `find`, sunt puse lock-uri pe nodurile la care thread-ul curent are acces la momentul respectiv, iar acestea sunt deblocate imediat ce nu mai este nevoie de ele. Aceasta maniera de blocare si deblocare permite un acces concurent la lista simplu inlantuita. Spre exemplu, in metoda de `find`, este efectuata o traversare a listei in cautarea unui anumit element, iar verificarea valorii unui nod implica blocarea acelui nod, urmata apoi de deblocarea acestuia. In metoda de add, in timpul parcurgerii listei pentru a gasi locul de adaugat monomul, thread-ul curent are `lock` la 2 noduri in fiecare moment – acestea vor fi nodurile intre care va fi adaugat monomul, dat ca si parametru. Dupa ce le termina de prelucrat, thread-ul le face `unlock`.

De asemenea, pentru a evita problemele cu schimbarea capului de lista, este folosit un semafor binar; semaforul este blocat inainte de a se accesa "head"-ul listei si deblocat cand nu mai este facuta nicio prelucrare a acestui nod. Astfel, se permite un acces exclusiv la lista, in materie de prelucrarea "head"-ului. Acest mecanism este necesar pentru a evita situatiile in care un thread citeste "head"-ul, iar altul il modifica la adaugare, deoarece monomul de adaugat are exponentul mai mic decat cel din "head".

`MyBlockingQueue` reprezinta o implementare proprie a conceptului de `BlockingQueue`, regasit in libraria standard din Java, unde operatia de scoatere a unui element dintr-o coada este blocanta, pana coada are un element care poate fi scos. `MyBlockingQueue` suporta doar operatiile necesare pentru aceasta tema de laborator, respectiv `add` si `poll`, ce respecta specificatia regasita in interfata `Queue` din libraria standard. Pentru a refolosi cod, `MyBlockingQueue` se foloseste de o instanta de `Queue` pentru a stoca elementele si implementeaza partea de sincronizare in metodele ei. Sincronizarea se realizeaza cu ajutorul mecanismului de `wait` / `notify` – atunci cand se adauga in coada, thread-ul curent notifica celelalte thread-uri, iar atunci cand un thread doreste sa scoata ceva din coada, daca aceasta este vida, asteapta. In plus, este folosit si mecanismul de `synchronized` caracteristic limbajului Java, pentru a permite accesul exclusive la coada.

Atat varianta secventiala, cat si cea paralela, se folosesc de toate clasele din pachetul `utils.io`, responsabile cu citirea si scrierea polinoamelor din fisiere.

Facand o analogie cu problema producatorului-consumatorului, metoda readParallel din `PolynomialReader` reprezinta producatorul, in timp ce metoda run din `utils.threading.AddParallel`, consumatorul. Spre deosebire de Laboratorul 4, se folosesc mai multi producatori. Acestia citesc cate un monom din fisierele date si il adauga in coada de tip `MyBlockingQueue`, ce va fi preluat de catre un thread ce executa metoda run, acesta primind aceeasi instanta de `MyBlockingQueue`. Dupa ce il citeste, il adauga la un polinom rezultat, initializat cu polinomul nul, care poate fi accesat de toate thread-urile, prin metoda `add` din clasa `Polynom`.

Dupa ce un producator termina de citit in intregime fisierele date, el va trimite unui anumit numar de thread-uri un monom invalid, cu exponentul egal cu -1. Numarul de thread-uri la care se vor trimite monoamele invalide de catre un producator este decis echitabil, in functie de numarul total de producatori. Thread-urile consumatoare vor detecta fiecare aparitia acestui monom cand scot un element din coada si isi vor incheia executia.

Cazuri de testare

1)

$f[X] = 1 + 2X + 3X^2 + X^7$

$g[X] = 4 + 2X - 9X^7$

$h[X] = 90 + 2X^9$

Rezultat asteptat: $95 + 4X + 3X^2 - 8X^7 + 2X^9$

2)

$f[X] = 55X + 47X^3 + 58X^4 + 84X^5$

$g[X] = 8X + 46X^2 + 32X^3 + 75X^5$

$h[X] = 21X + 61X^2 + 46X^3 + 57X^4$

$i[X] = 72X - 125X^3 + 90X^4$

Rezultat asteptat: $156X + 107X^2 + 205X^4 + 159X^5$

Rezultate

Tip polinoame	**Numar thread-uri**	**Timp executie (ms)**
10 polinoame, fiecare cu gradul maxim 1000 si cu maxim 100 monoame *(test1)*	secvential	127
	P1=2, P2=2	135
	P1=3, P2=1	140
	P1=2, P2=4	129
	P1=3, P2=3	137
	P1=2, P2=6	139
	P1=3, P2=5	135
5 polinoame, fiecare cu gradul maxim 10000 si cu maxim 500 monoame *(test2)*	secvential	182
	P1=2, P2=2	242
	P1=3, P2=1	180
	P1=2, P2=4	245
	P1=3, P2=3	240
	P1=2, P2=6	236
	P1=3, P2=5	263

Nota: fiecare test a fost rulat de 10 ori si pentru evaluarea timpului de executie s-a considerat media aritmetica a acestor 10 rulari. Fiecare test a fost

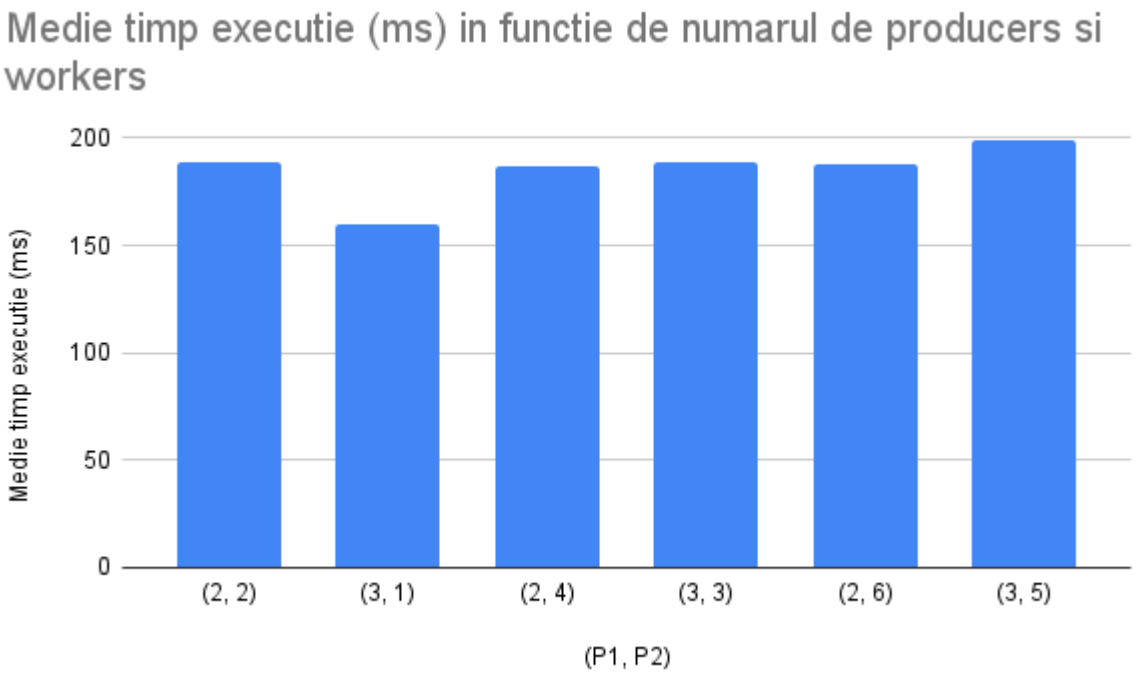
rulat pe o masina cu Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 MHz 2 cores 4 Logical Processors cu sistem de operare Windows 10 versiunea 21H1.

Analiza rezultatelor

Se poate observa ca varianta secventiala este per total cea mai performanta. Un posibil motiv este faptul ca este destul de costisitoare intretinerea thread-urilor si executia multor blocari si deblocari, care sunt necesare pentru a evita *race condition*, aducand astfel un overhead destul de mare, dar si faptul ca masina pe care au fost rulate testele are doar 2 nuclee.

Dintre variantele paralele, se observa ca, in medie, varianta in care se foloseste un singur thread consumator este cea mai performanta. Un bun motiv este faptul ca, fiind un singur thread, sincronizarile nu au, in esenta, niciun efect.

(P1, P2)	Medie timp executie (ms)
(2, 2)	188.5
(3, 1)	160
(2, 4)	187
(3, 3)	188.5
(2, 6)	187.5
(3, 5)	199



In comparatie cu Laboratorul 4, se poate observa ca acesta este mai performant decat Laboratorul 5, un motiv fiind overhead-ul indus de operatiile de blocare si deblocare.

In cadrul ambelor laboratore se observa faptul ca timpii medii de executie cresc direct proportional cu numarul de thread-uri folosite.

Numar thread-uri	Medie timp de executie (ms) Lab4	Medie timp de executie (ms) Lab5
4	158	174.25
6	159	187.75
8	164	193.25