PPD - Tema 2

Timpi de executie

Tip matrice	**Nr. threads**	**Timpi (ms)**	
		Java	**C++**
N=M=10, n=m=3	Sequencial	0.5ms	0.00394ms
	4	0.3ms	1.1763ms
N=M=1000, n=m=3	Sequencial	128.0ms	19.74545ms
	2	150.4ms	16.24848ms
	4	120.9ms	9.55517ms
	8	104.1ms	9.9424ms
	16	129.4ms	10.5809ms
N=M=10000, n=m=3	Sequencial	2668.9ms	2533.73ms
	2	2052.1ms	1699.974ms
	4	2456.6ms	989.4704ms
	8	2724.2ms	975.3337ms
	16	2345.4ms	993.2761ms

Observatii

Tip Matrice	Java	C++
N=M=10	Varianta paralela se intampla sa fie cu 20% mai rapida decat cea secventiala	Varianta secventiala este “de sute de ori” mai rapida decat cea paralela. (irrelevant totusi deoarece datele sunt de dimensiuni mici, este posibil ca lucrul cu threaduri sa coste mai mult decat convolutia in sine)
N=M=1000	Se observa imbunatatiri in folosirea a 8 threaduri (18.67%). Varianta cu 4 threaduri este putin mai lenta, iar cea cu 16 threaduri devine relativ ineficienta. Pentru 2 threaduri, avem o intarziere de 17.5%	Toate variantele paralele intrec performanta celei secventiale: p=2 (17.73%), p=4 (51%), p=8 (49%), p=16 (46%). Variantele cu 4 si 8 threaduri sunt cele mai rapide, necesitand apropae jumatate din timpul de la secv.
N=M=10000	Varianta cu 2 threaduri este cea mai rapida (23%), urmata de p=16 (12%), p=4 (7%). Pentru p=8 observem un timp mediu mai mare decat secventialul	Se observa o dependent invers proportional intre nr. de threaduri si timpul de executie. Daca 2 threaduri aproape au redus timpul de executie cu 32%, folosirea a 4 threaduri devince cu inca 41% mai eficienta, pentru ca apoi impactul a 8 threaduri sa nu mai fie atat de puternic (<1%). Folosirea a 16 threaduri deja se dovedeste a fi mai putin eficienta

Varianta in C++ este in general mai rapida decat implementarea in Java (exceptie N=10, p=4), cu un factor de 10 (N=1000), respectiv de 1-3 (N=10000). Timpii medii sunt foarte apropiati in cazul unui volum mare de date procesat secvential (N=10000), dar acestia diverg rapid daca se include folosirea threadurilor.

Explicarea algoritmului

Descriem procedura de efectuare a convolutiei $conv(K, A, i_0, i_1)$ cu nucleul K in-place pentru submatricea determinata de liniile $i_0, i_0 + 1, \dots, i_1 - 1$ a matricei A. Pentru variant secventiala, se va aplica $conv(K, A, 0, N)$. Pentru $p \geq 2$ threaduri, se impart liniile in p subsecvente $\left(i_0^k, i_1^k\right), k = \overline{1,p}$ care respecta distributia liniara echitabila (ex. formarea unor subsecvente initiale de cate $\left[\frac{N}{p}\right]$ linii, si redistributirea restului de $N\%p$ linii, cate una, grupurilor formate, pana cand numarul lor se epuizeaza), si se aplica $conv(K, A, i_0^k, i_1^k)$ pentru fiecare $k = \overline{1,p}$.

Ne concentram acum pe descrierea functiei $conv$.

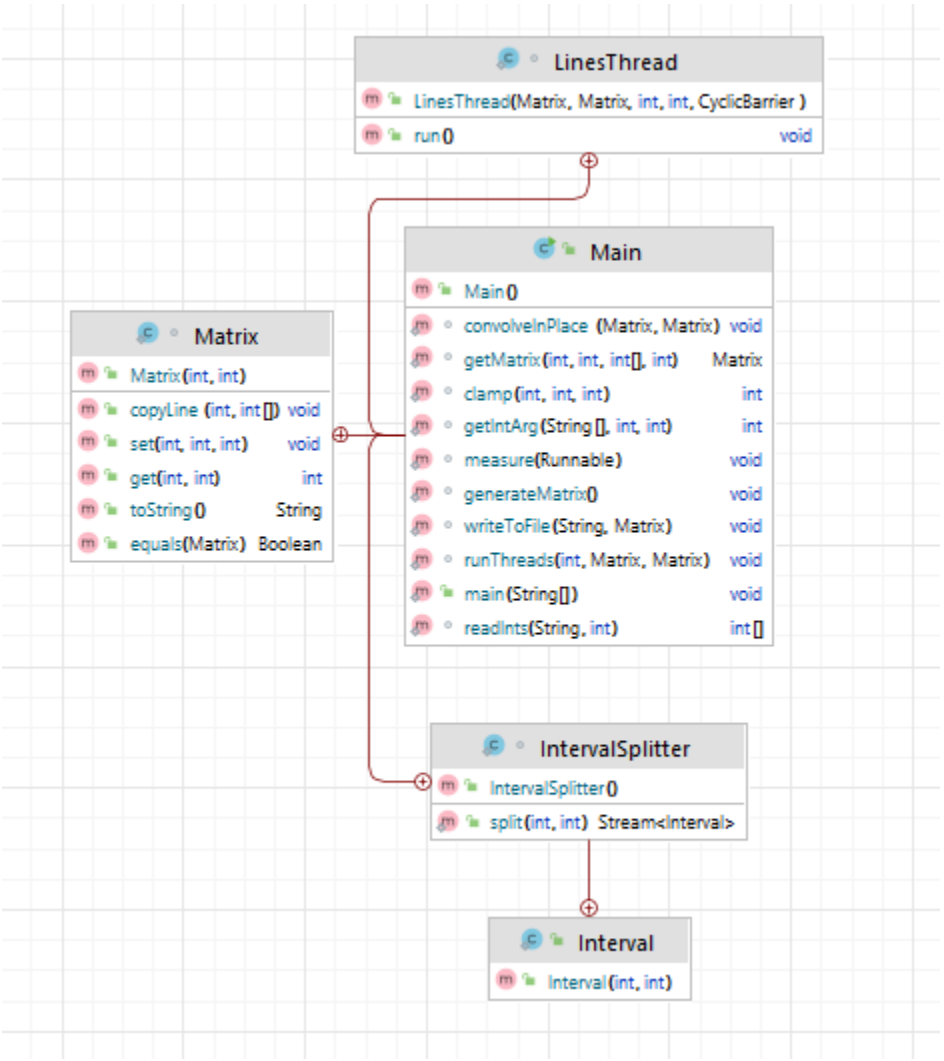
Observam ca pentru a aplica convolutia de nucleu $K \in \mathcal{M}_{3,3}$ asupra unui element $A_{i,j}$, sunt implicate 3 linii: linia anterioara $i - 1$ (sau 0, in cazul bordarii), linia curenta i, si linia urmatoare $i + 1$ (sau $N - 1$, in cazul bordarii). Conflictele de memorie apar intre doua functii "consecutive" $c1 = conv(K, A, i_0, i_1)$ si $c2 = conv(K, A, i_1, i_2)$ in doua locuri: in $c1$ la linia $i_1 - 1$ (pentru ca linia i_1 apartine de $c2$ si poate fi suprascrisa $(*)$), respectiv in $c2$ la linia i_1 (pentru ca linia $i_1 - 1$ apartine de $c1$). Aceste linii "problema" trebuie memorate intr-un buffer separate al fiecarui thread inainte de a executa orice operatii. Liniile "curente" vor fi intotdeauna in zona de actiune a fiecarui thread, insa si acestea sunt supuse modificarii chiar in interiorul threadului insusi: pentru a calcula $A_{i,j}$ avem nevoie de $A_{i,j-1}$, posibil suprascris anterior. De aceea, se impune inclusiv memorarea liniei curente.

Pseudocod:

```
func conv(K,A,i0,i1)
    prevLine = copy(A[max(i0-1,0)])
    lastLine = copy(A[min(i1, N-1)])
    // in cazul threadurilor, barrier.wait() aici
    for i in i0…i1-1 do
        crtLine = copy(A[i])
        nextLine = i==i1—1 ? lastLine : A[i+1] // referinta, nu copy
        for j in 0…M-1 do
            A[i][j] = atomic_conv(K, j,prevLine, crtLine, nextLine)
        endfor
        prevLine = crtLine
    endfor
endfunc

func atomic_convK, (j, prevLine, crtLine, nextLine)
   return K[0][0]*prevLine[j-1] + K[0][1]*prevLine[j] + K[0][1]*prevLine[j+1]
         +K[1][0]* crtLine[j-1] + K[1][1]* crtLine[j] + K[1][1]* crtLine[j+1]
         +K[2][0]*nextLine[j-1] + K[2][1]*nextLine[j] + K[2][1]*nextLine[j+1]
   // prin crtLine[j] se intelege crtLine[clamp(j,0,M-1)] !
endfunc
```

Complexitatea memoriei de lucru: Mem(P,N,M) = 3*P*M => O(P*M), P=nr. threaduri, N,M=dim. matricei


LinesThread
LinesThread(Matrix, Matrix, int, int, CyclicBarrier)
run() void
Main
Main()
convolveInPlace (Matrix, Matrix) void
getMatrix(int, int, int[], int) Matrix
clamp(int, int, int) int
getIntArg(String [], int, int) int
measure(Runnable) void
generateMatrix() void
writeToFile(String, Matrix) void
runThreads(int, Matrix, Matrix) void
main(String[]) void
readInts(String, int) int[]
Matrix
Matrix(int, int)
copyLine (int, int[]) void
set(int, int, int) void
get(int, int) int
toString() String
equals(Matrix) Boolean
IntervalSplitter
IntervalSplitter()
split(int, int) Stream<Interval>
Interval
Interval(int, int)