Imię i nazwisko: Anastasiya Yahorava, Marcin Wardyński
Grupa: Piątek, 15:00

Sprawozdanie do programu mnożenia macierzy

Wariant programu:

Dla macierzy o rozmiarze mniejszym lub równym 2^l × 2^l algorytm tradycyjny.
Dla macierzy o rozmiarze większym od 2^l × 2^l algorytm rekurencyjny Binéta.

Rozmiar macierzy użytej w zadaniu: 2^10x2^10

Kod programu:

https://github.com/mwardynski/martrix-calculus/tree/prog1/prog1

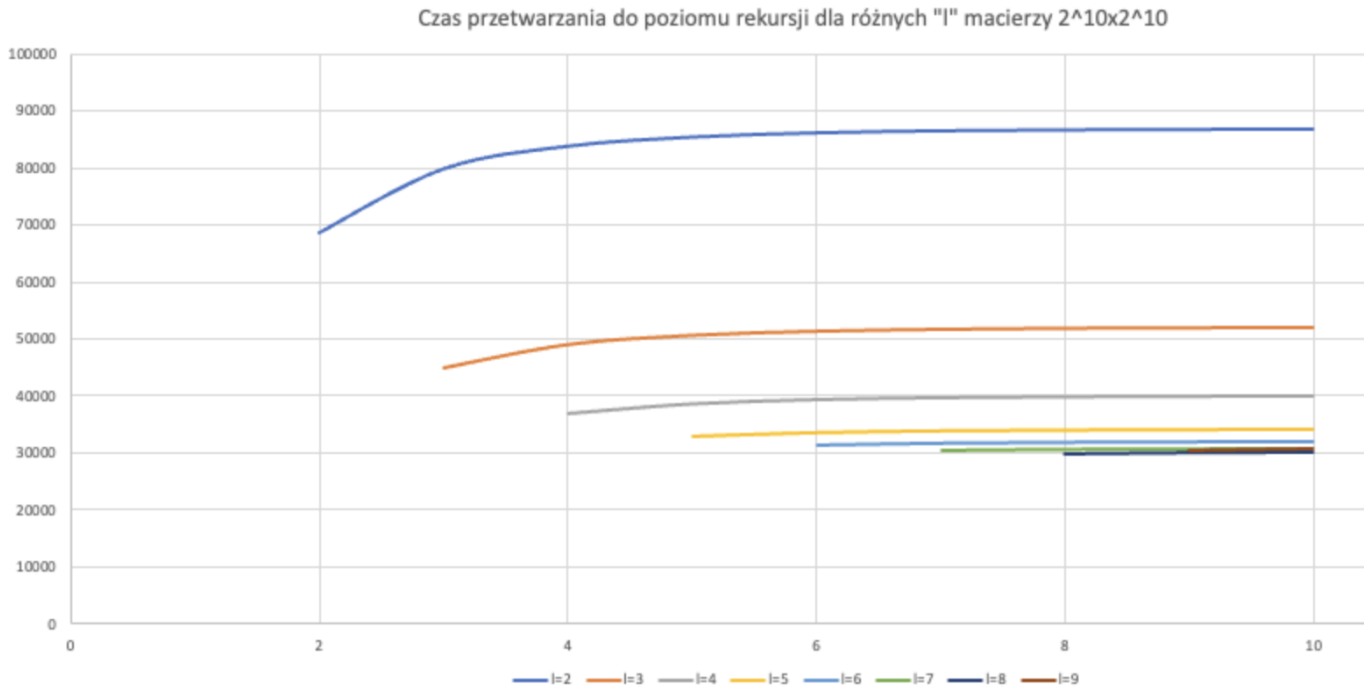
1. Pseudokod

```
tradMultiply(A, B) {
    AB = createMatrix(A.length, A[0].length)
    for (i in A) {
        for (j in B[0]) {
            sum = 0
            for (k in A[0]) {
                sum += A[i][k]*B[k][j]
            }
            AB[i][j] = sum
        }
    }
    return AB;
}

binetMultiply(A, B, l, l_switch) {
    if(l == l_switch) {
        return tradMultiply(A, B)
    }
    else {
        qsA = splitToQuarters(A)
        qsB = splitToQuarters(B)

        resultQs = createMatrix(2, 2)
        resultQs[0][0] = addMatrices(
            binetMultiply(qsA[0][0], qsB[0][0], l-1, l_switch),
            binetMultiply(qsA[0][1], qsB[1][0], l-1, l_switch))
        resultQs[0][1] = addMatrices(
            binetMultiply(qsA[0][0], qsB[0][1], l-1, l_switch),
            binetMultiply(qsA[0][1], qsB[1][1], l-1, l_switch))
        resultQs[1][0] = addMatrices(
            binetMultiply(qsA[1][0], qsB[0][0], l-1, l_switch),
            binetMultiply(qsA[1][1], qsB[1][0], l-1, l_switch))
```

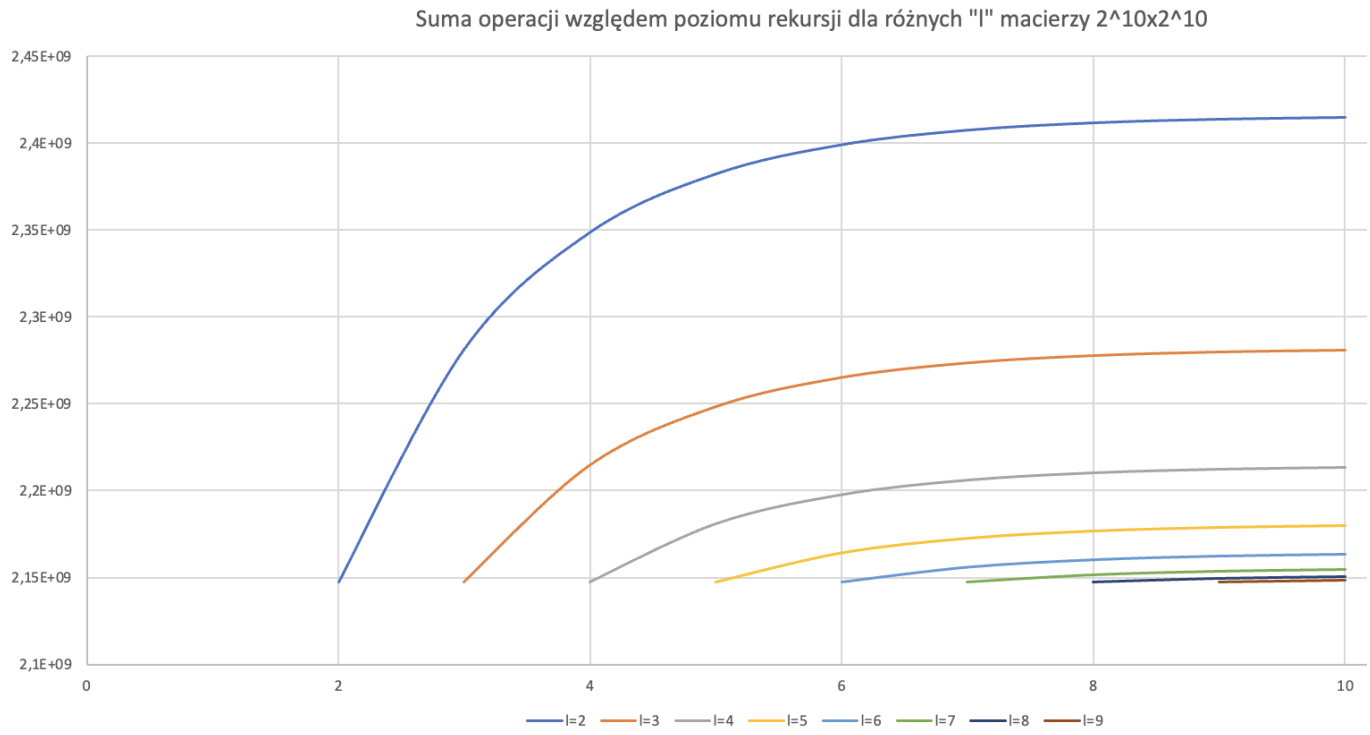
```
            resultQs[1][1] = addMatrices(
                binetMultiply(qsA[1][0], qsB[0][1], l-1, l_switch),
                binetMultiply(qsA[1][1], qsB[1][1], l-1, l_switch))

            return flattenQuarters(resultQs)
        }
    }
```

Krótki opis użytych funkcji pomocniczych:
createMatrix - tworzy macierz o rozmiarze zadanym w parametrach. *splitToQuarters* - dzieli podaną macierz na ćwiartki, przykładowo z jednej macierzy 4x4 powstaną cztery macierze 2x2. *addMatrices* - dodaje dwie macierze do siebie. *flattenQuarters* - operacja odwrotna do *splitToQuarters*, która traktuje podane cztery macierze jako ćwiartki i spłaszcza je do jednej macierzy, w ten sposób przykładowo z czterech ćwiartek 2x2 otrzymujemy macierz 4x4.

2. Wykres czasu przetwarzania w stosuknu do poziomu rekursji dla różnych "l"



3. Wykres ilości operacji zmiennoprzecinkowych w stosuknu do poziomu rekursji dla różnych "l"



Interpretacja wyników:

Na wykresach została zobrazowana wydajność mnożenia macierzy przy użyciu tradycyjnego algorytmu mnożenia i rekurencyjnego mnożenia Binet-a.

W przypadku stosowania tradycyjnego mnożenia na najgłębszym poziomie rekursji, liczba operacji może być bardzo duża, co prowadzi do zwiększonego czasu przetwarzania. Jednak po zmniejszeniu poziomu rekursji stosującego mnożenie tradycyjne aż do poziomu l = 5, zaobserwowano znaczące obniżenie wymaganych operacji i skrócenie czasu przetwarzania. Dalsze poziomy nie wykazują już tak wyraźnej redukcji operacji i czasu

Analizując podane wyniki, możemy zauważyć, że im wyższy jest poziom rekursji stosującej mnożenie tradycyjne, tym mniejsza jest liczba operacji, co przekłada się na mniejszy czas przetwarzania. Oznacza to, że głębokość rekursji ma istotny wpływ na wydajność algorytmu. Na płytkich poziomach rekursji wzrost czasu przetwarzania i liczby operacji nie wpływa na działanie algorytmu, ale około od poziomu l=5 i niżej widoczne już są znaczne zmiany i podział na bloki staje się nieskuteczny dla przetwarzania sekwencyjnego.

Główną przyczyną powyższych wniosków jest stała ilość operacji potrzebnych dla mnożenia tradycyjnego. Nieważne, czy mnożenie to nastąpiło dla l=2, czy l=8, mnożenie tradycyjne pokryje całą macierz wykonując ok. 2,14E+7 operacji. Jeśli mnożenie tradycyjne zostało użyte głęboko w rekursji, to do wyniku tego dojdą jeszcze operacje scalania dla każdego z poziomów rekursji. Ergo, mnożenie blokowe jest wolniejsze od mnożenia tradycyjnego dla przetwarzania sekwencyjnego i daje poprawę dopiero przy zrównolegleniu, gdzie poszczególne bloki zostałyby rozdystrybuowane do rozproszonych jednostek obliczeniowych.