Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра математики факультету інформатики

Реалізація рекурсивно-блочного алгоритму LDU факторизації матриці з використанням суперкомп'ютера

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні Науки"

Керівник курсової роботи
ас. Сідько А.А.

_____
(підпис)
"____" _____ 2022 р.

Виконав студент 4 курсу
Комп'ютерні Науки
Березніков О.Д.
"____" _____ 2022 р.

Київ 2022

Тема: LDU факторизація матриць з використанням суперкомп’ютера

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	03.10.2022	
2.	Огляд технічної літератури за темою роботи.	15.11.2022	
3.	Аналіз паралельного алгоритму LDU факторизація матриць з використанням суперкомп’ютера.	25.11.2022	
3.	Розробка паралельної динамічної програми LDU факторизації матриці з використанням стандартних механізмів Java.	30.12.2022	
4.	Реалізація розробленої програми.	15.01.2022	
5.	Тестування розробленої програми.	25.01.2022	
6.	Аналіз тестування та виправлення помилок.	30.01.2022	
7.	Проведення експериментів	03.02.2022	
8.	Розробка паралельної динамічної програми LDU факторизації матриці з використанням бібліотеки DAP.	10.02.2022	
9.	Реалізація розробленої програми.	01.03.2022	
10.	Тестування розробленої програми.	30.03.2022	
11.	Аналіз тестування та виправлення помилок.	15.04.2022	

12.	Проведення експериментів	18.04.2022	
13.	Написання пояснювальної роботи.	05.05.2022	
14.	Аналіз отриманих результатів з керівником.	10.05.2012	
10.	Коригування роботи за результатами попереднього захисту.	15.05.2022	
11.	Створення слайдів та написання доповіді.	26.05.2022	

Студент Березніков О.Д.
Керівник Сідько А. А.
“_____” _____

Зміст

Анотація 5

Вступ 6

Розділ 1: Опис блочно-рекурсивного алгоритму LDU(класичний LU), та його рекурсивної модернізації. 8

1.1 Опис класичного LU алгоритму 8

1.2 Опис LDU алгоритму 9

1.3 Опис LDU алгоритму з рекурсивною модернізацією [1] 9

1.4 Зведення всіх формул рекурсивно-блочного алгоритму LDU [1] 13

Розділ 2: Опис реалізації багатопоточної рекурсивної модернізації за допомогою засобів Java для роботи з потоками. 16

2.2.1 Паралельна LDUMW факторизація матриці засобами Java 18

2.2.2 Багатопоточне рекурсивне множення використане в розпаралеленому алгоритмі 23

Розділ 3: Опис DAP системи, динамічної системи розрахунків, кроків алгоритму, та основним складовим цієї системи. 26

3.1 Загальний опис основних дій для використання системи DAP 26

Розділ 4: Реалізація алгоритму на системі DAP 28

4.1 Використання спеціального об’єкта для зберігання інформації 28

4.2 Основний дроп для LDU факторизації - LdumwFact 28

4.2.1 Визначення арків 28

4.2.1 Визначення конструктора 31

4.2.2 Визначення методу doAmin 32

4.2.3 Визначення методу sequentialCalc 33

4.2.4 Визначення методу inputFunction 34

4.2.5 Визначення методу outputFunction 35

4.3 Дроп для множення LDU факторизації 35

4.4 Клас для тестування дропу LDUMW факторизації 39

Розділ 5: Проведення експериментів розроблених алгоритмів на системі DAP та за допомогою засобів Java. 40

5.1 Проведення експериментів на багатопотоковій модернізації розробленій засобами Java 41

5.2 Проведення експериментів на багатопотоковій модернізації розробленій на системі DAP 43

Висновки 49

Список використаних джерел 50

Анотація

У роботі розглянуто рекурсивно-блочний алгоритм LDU факторизації матриці, для знаходження відповідних L - нижня трикутна матриця, U - верхня трикутна матриця, D - діагональна матриця. Основну увагу приділено створення децентралізованого алгоритму управління розподіленими обчисленнями, для розв’язання такого типу задач. В результаті дослідження було розроблено паралельну програму на OpenMPI, мовою програмування Java, з використанням пакету mathpar. Проведено серію експериментів та зроблено висновки щодо роботи програми та вдосконалення.

Ключові слова: алгоритм, паралельне програмування, процесор, матриця, граф, блочний алгоритм, множення, обернення, транспонування, дроп, амін, рекурсія, вхідні, вихідні дані, повідомлення, потік, список, результат.

Вступ

На сьогодні суперкомп’ютери є важливою складовою в різних сферах, таких як бізнес, медицина, військова справа і т.д. Суперкомп'ютер - це комп'ютер, ціль якого працювати на рівні або близькому до найвищої швидкості серед усіх комп’ютерів.

Традиційно суперкомп’ютери використовувалися для наукових та інженерних додатків, які повинні обробляти масивні бази даних, виконувати велику кількість обчислень або і те, і інше. Такі досягнення, як багатоядерні процесори та графічні процесори загального призначення, дозволили створити потужні машини, які можна було б назвати настільними суперкомп'ютерами або суперкомп'ютерами з графічним процесором.

За визначенням, суперкомп’ютер є винятковим з точки зору продуктивності. У будь-який час існує кілька широко розголошених суперкомп’ютерів, які працюють на надзвичайно високої швидкості порівняно з усіма іншими комп’ютерами. Термін суперкомп'ютер іноді застосовується до набагато повільніших, але все одно вражаюче швидких комп'ютерів.

Об’єктом дослідження курсової роботи є рекурсивно-блочний алгоритм LDU факторизації матриці для знаходження нижньої трикутної матриці, верхньої трикутної матриці та діагональної матриці. Програма може бути написана з використанням багатопоточності, тому-що існує можливість виконувати кроки алгоритму окремо один від одного, що призведе до лінійного прискорення програми.

Мета роботи полягає в розробці багатопотокової програми яка б виконувалась за час, який повинен бути менший за виконання послідовного аналогу. Також метою є проведення дослідження, у якому треба з’ясувати, чи є даний багатопотоковий алгоритм факторизації матриці ефективніший за паралельний.

У першому розділі розглянуто загальний блочно-рекурсивний алгоритм LDU(класичний LU), та його рекурсивна модернізація.

Другий розділ описує реалізацію багатопоточної рекурсивної модернізації за допомогою засобів Java для роботи з потоками.

Третій розділ присвячений короткому опису DAP системи, динамічній системі розрахунків, крокам алгоритму, та основним складовим цієї системи, які мусять бути запрограмовані.

В четвертому розділі описується реалізація алгоритму на системі DAP та детальний опис коду.

Останній розділ містить проведення експериментів розроблених алгоритмів на системі DAP та за допомогою засобів Java. Знаходження оптимальних листових вершин.

Розділ 1: Опис блочно-рекурсивного алгоритму LDU(класичний LU), та його рекурсивної модернізації.

1.1 Опис класичного LU алгоритму

Подання матриці A у вигляді двох факторів A = LU, де L - нижня трикутна матриця, а U - верхня трикутна матриця, називається LU декомпозиція (або LU факторизація). Це розкладання розглядається як базовий алгоритм для будь-якої бібліотеки програм лінійної алгебри. Багато алгоритмів на її основі будуються, в тому числі розв'язування лінійних систем, обернення матриць, ранг розрахунок тощо. недоліки для полів скінченних чисел. Це був перший блочно-рекурсивний алгоритм зі складністю матричного множення, розрідженим і паралельним. Проте проблема втрати точності у випадку класичних полів залишалася як і раніше нерозв'язною. Потрібно було отримати узагальнення факторизації LEU на комутативні області та їх поля часткових. Такий алгоритм дозволить, наприклад, використовувати в обчисленнях цілі числа замість наближених раціональних чисел. Було запропоновано два підходи до створення такий алгоритм факторизації LDU.

Основні кроки, необхідні для розв’язання системи рівнянь за допомогою розкладання LU, є такими. Оскільки $\mathbf{A} = \mathbf{LU}$, то $\mathbf{Ax} = \mathbf{b}$ стає $\mathbf{LUx} = \mathbf{b}$, де b не обмежується одним стовпцем. Дозволяє $\mathbf{y} = \mathbf{Ux}$, що веде до $\mathbf{Ly} = \mathbf{b}$.

Оскільки L є нижньою трикутною матрицею, це рівняння ефективно розв’язується шляхом прямої заміни. Щоб знайти x, ми розв’язуємо

$$\mathbf{Ux} = \mathbf{y}$$

Оскільки U є верхньою трикутною матрицею, це рівняння також можна ефективно розв’язати шляхом зворотної заміни.

1.2 Опис LDU алгоритму

Хоча це не має нічого спільного з розв'язуванням систем, A = LU не є "симетричним" в тому сенсі, що L має одиниці на головній діагоналі, а U – ні. Іноді ми змінюємо U, витягнувши з кожного рядка. Нагадаємо, що поділити рядок на ненульовий запис це елементарна операція. Для цього ми множимо U зліва на матрицю зліва для множення кожного рядка за номером. Тобто ми замінюємо LU на LDU, де D — діагональна матриця центрів, і ця нова U є старою U, кожен рядок якої поділений на множник. Тобто U замінена на DU. Факторізації A = LDU полягає в тому, що вона унікальна.

1.3 Опис LDU алгоритму з рекурсивною модернізацією [1]

$$(L, D, U, M, \widehat{D}, W, \alpha_r) = \mathbf{LDU}(A, \alpha)$$

LDU-факторизація рекурсивним і дихотомічним способом.

(1) Якщо $A = 0$, то будемо вважати, що $D = I = J = 0,\ M = W = \alpha\mathbf{I},\ \widehat{D} = \alpha^{-1}\mathbf{I},$
$L = U = \bar{I} = \bar{J} = \bar{D} = \mathbf{I},\ \alpha_r = \alpha.$

(2) Якщо $n = 1\ (A = [a], a \neq 0)$, то будемо вважати, що
$D = [(\alpha a)^{-1}],\ \widehat{D} = [a^{-2}],\ L = U = M = W = [a],\ I = J = [1],$
$\bar{I} = \bar{J} = \bar{D} = [0],\ \alpha_r = a.$

(3) Для $A \neq 0$ та n > 1 ми ділимо матрицю A на чотири частини

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

Ми можемо зробити LDU-декомпозицію для блоку A_{11}:

$$\begin{cases} \alpha L_{11} D_{11} U_{11} = A_{11} \\ L_{11} \widehat{D}_{11} M_{11} = I \\ W_{11} \widehat{D}_{11} U_{11} = I \end{cases}.$$

Нехай k = rank(A_{11}) та det_1, det_2,...,det_k є ненульовими вкладеними провідними мінорами з A_{11}, які були знайдені рекурсивно, $\alpha_k = det_k$, $\widehat{D}_{11} = \alpha_k^{-1}(\alpha D_{11} + \bar{D}_{11})$ та ненульові елементи D_{11} рівні $(\alpha det_1)^{-1}$, $(det_1 det_2)^{-1}$ ... $(det_{k-1}\alpha_k)^{-1}$. Після цього ми можемо створити рівняння:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \alpha D_{11} & \frac{1}{\alpha_k} A'_{12} \\ \frac{1}{\alpha_k} A'_{21} & A_{22} \end{pmatrix} \begin{pmatrix} U_{11} & 0 \\ 0 & I \end{pmatrix}.$$

Позначимо нові блоки:

$$A'_{12} = \alpha_k \widehat{D}_{11} M_{11} A_{12}, \quad A'_{21} = \alpha_k A_{21} W_{11} \widehat{D}_{11}.$$

Які можна декомпозувати в:

$$\begin{pmatrix} \alpha D_{11} & \frac{1}{\alpha_k} A'_{12} \\ \frac{1}{\alpha_k} A'_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ \frac{1}{\alpha} A'_{21} D_{11}^{+} & I \end{pmatrix} \begin{pmatrix} \alpha D_{11} & \frac{\alpha}{\alpha_k} A''_{12} \\ \frac{\alpha}{\alpha_k} A''_{21} & \frac{1}{\alpha_k} A'_{22} \end{pmatrix} \begin{pmatrix} I & \frac{1}{\alpha} D_{11}^{+} A'_{12} \\ 0 & I \end{pmatrix}.$$

Позначаємо

$$E_{11} = D_{11}^{\rightarrow 1}, \quad I_{11} = E_{11} E_{11}^{T}, \quad J_{11} = E_{11}^{T} E_{11}.$$

Для позначення узагальненої оберненої матриці використовуємо верхній індекс плюс. Для будь-якої матриці $(a_{i,j}) \in \mathbf{S}_{wp}$ узагальнена обернена матриця збігається з псевдозворотньою матрицею:

$$(a_{i,j})^{+} = \{(b_{i,j}) : b_{i,j} = 0 \text{ if } a_{j,i} = 0,\ b_{i,j} = a_{j,i}^{-1} \text{ if } a_{j,i} \neq 0\}.$$

Псевдоінверсна матриця виходить шляхом транспонування заданої матриці та заміни всіх ненульових елементів з оберненими елементами.

Так як $\bar{I}_{11}\widehat{D}_{11} = \bar{D}_{11} = \widehat{D}_{11}\bar{J}_{11}$, $D_{11}^{+}\bar{I}_{11} = 0$ і $D_{11}^{+}\widehat{D}_{11} = J_{11}$ ми отримуємо:

$$A''_{12} = \frac{1}{(\alpha\alpha_k)} \bar{I}_{11} A'_{12} = \frac{1}{\alpha} \bar{D}_{11} M_{11} A_{12}, \ A''_{21} = \frac{1}{(\alpha\alpha_k)} A'_{21} \bar{J}_{11} = \frac{1}{\alpha} A_{21} W_{11} \bar{D}_{11}.$$

Для нижнього правого блоку ми отримуємо:

$$\alpha\alpha_k^2 A_{22} = (A'_{21}J_{11} + A''_{21})D_{11}^{+}A'_{12} + A'_{21}D_{11}^{+}A''_{12} + \alpha\alpha_k A'_{22}$$

$$A'_{22} = \frac{1}{\alpha\alpha_k}(\alpha\alpha_k^2 A_{22} - A'_{21}D_{11}^{+}A'_{12}).$$

Матриці A''_{12} та A''_{21} є матрицями оточуючих мінорних щодо мінорної αk.

Нехай $\begin{cases} \alpha_k L_{21}D_{21}U_{21} = A''_{21} \\ L_{21}d_{21}M_{21} = I \\ W_{21}d_{21}U_{21} = I \end{cases}$ та $\begin{cases} \alpha_k L_{12}D_{12}U_{12} = A''_{12} \\ L_{12}d_{12}M_{12} = I \\ W_{12}d_{12}U_{12} = I \end{cases}$

є LDU декомпозиція для A''_{12} та A''_{21}

Позначимо

$$\alpha_l = det_l,\ \alpha_s = det_s,\ J_{12}^{\lambda} = \lambda J_{12} + \bar{J}_{12},\ I_{12}^{\lambda} = \lambda I_{12} + \bar{I}_{12},$$

$$L_{12}^{\sim} = L_{12}I_{12}^{\lambda},\quad U_{12}^{\sim} = J_{12}^{\lambda}U_{12},\quad D_{12}^{\sim} = \lambda^{-2}D_{12},$$

$$\widehat{D}_{12}^{\sim} = \lambda^{-1}I_{12}^{\lambda^{-1}}\widehat{D}_{12},\quad M_{12}^{\sim} = \lambda M_{12},\quad W_{12}^{\sim} = \lambda W_{12},$$

Остання система рівнянь:

$$\begin{cases} (\lambda\alpha_k)L_{12}^{\sim}D_{12}^{\sim}U_{12}^{\sim} = \lambda A''_{12} \\ L_{12}^{\sim}\widehat{D}_{12}^{\sim}M_{12}^{\sim} = I \\ W_{12}^{\sim}\widehat{D}_{12}^{\sim}U_{12}^{\sim} = I \end{cases}.$$

З цієї системи можна позначити:

$$\begin{pmatrix} \alpha D_{11} & \frac{\alpha}{\alpha_k}A''_{12} \\ \frac{\alpha}{\alpha_k}A''_{21} & \frac{1}{\alpha_k}A'_{22} \end{pmatrix} = \begin{pmatrix} L_{12}^{\sim} & 0 \\ 0 & L_{21} \end{pmatrix}\begin{pmatrix} \alpha D_{11} & \alpha D_{12}^{\sim} \\ \alpha D_{21} & \frac{1}{\alpha_k}A''_{22} \end{pmatrix}\begin{pmatrix} U_{21} & 0 \\ 0 & U_{12}^{\sim} \end{pmatrix},$$

де ми виводимо:

$$A''_{22} = \widehat{D}_{21}M_{21}A'_{22}W_{12}^{\sim}\widehat{D}_{12}^{\sim} = \widehat{D}_{21}M_{21}A'_{22}W_{12}I_{12}^{\lambda^{-1}}\widehat{D}_{12}$$

Виводимо з поточного рівняння

$$I_{12}I_{11} = 0,\ J_{11}J_{21}$$

$$L_{12}D_{11}U_{21} = D_{11}.$$

Приводимо до нормального вигляду:

$$L_{12} = L_{12}I_{12} + \bar{I}_{12},$$

$$D_{11} = I_{11}D_{11}J_{11},$$

$$U_{21} = J_{21}U_{21} + \bar{J}_{21}$$

Середня матриця може бути зведена до такого вигляду:

$$\begin{pmatrix} \alpha D_{11} & \alpha D_{12}^{\sim} \\ \alpha D_{21} & \frac{1}{\alpha_k} A_{22}'' \end{pmatrix} = \begin{pmatrix} I & 0 \\ \frac{1}{\alpha\alpha_k} A_{22}'' D_{12}^{\sim +} & I \end{pmatrix} \begin{pmatrix} \alpha D_{11} & \alpha D_{12}^{\sim} \\ \alpha D_{21} & \frac{\alpha}{\alpha_s} A_{22}''' \end{pmatrix} \begin{pmatrix} I & \frac{1}{\alpha\alpha_k} D_{21}^{+} A_{22}'' \bar{J}_{12} \\ 0 & I \end{pmatrix}$$

З якої ми можемо вивести

$$A_{22}''' = \frac{\alpha_s}{\alpha\alpha_k} \bar{I}_{21} A_{22}'' \bar{J}_{12} = \frac{1}{\alpha_k^2 \alpha} \bar{D}_{21} M_{21} A_{22}' W_{12} \bar{D}_{12}.$$

З вищеперерахованих пунктів ми можемо вивести L, D та U.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \alpha LDU, \quad D = \begin{pmatrix} D_{11} & D_{12}^{\sim} \\ D_{21} & D_{22} \end{pmatrix} = \begin{pmatrix} D_{11} & \lambda^{-2} D_{12} \\ D_{21} & D_{22} \end{pmatrix},$$

де $D_{12}^{\sim} = \lambda^{-2} D_{12}$.

$$L = \begin{pmatrix} L_{11} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ \alpha_k^{-1} A_{21}' D_{11}^{+} & I \end{pmatrix} \begin{pmatrix} L_{12}^{\sim} & 0 \\ 0 & L_{21} \end{pmatrix} \begin{pmatrix} I & 0 \\ \alpha^{-1}\alpha_k^{-1} \bar{I}_{21} A_{22}'' D_{12}^{\sim +} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & L_{22} \end{pmatrix},$$

$$U = \begin{pmatrix} I & 0 \\ 0 & U_{22} \end{pmatrix} \begin{pmatrix} I & \alpha^{-1}\alpha_k^{-1} D_{21}^{+} A_{22}'' \\ 0 & I \end{pmatrix} \begin{pmatrix} U_{21} & 0 \\ 0 & U_{12}^{\sim} \end{pmatrix} \begin{pmatrix} I & \alpha_k^{-1} D_{11}^{+} A_{12}' \\ 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & 0 \\ 0 & I \end{pmatrix}.$$

Після множення матриць ми отримуємо:

$$L = \begin{pmatrix} L_1 & 0 \\ L_3 & L_4 \end{pmatrix} = \begin{pmatrix} L_{11} L_{12}^{\sim} & 0 \\ L_3 & L_{21} L_{22} \end{pmatrix}, \quad U = \begin{pmatrix} U_1 & U_2 \\ 0 & U_4 \end{pmatrix} = \begin{pmatrix} U_{21} U_{11} & U_2 \\ 0 & U_{22} U_{12}^{\sim} \end{pmatrix},$$

де

$$U_2 = \alpha_k^{-1} U_{21} D_{11}^{+} A'_{12} + \alpha^{-1}\alpha_k^{-1} D_{21}^{+} A''_{22} U_{12}^{\sim} =$$
$$= \alpha_k^{-1} J_{11} M_{11} A_{12} + \alpha^{-1}\alpha_l^{-1} J_{21} M_{21} A'_{22},$$
$$L_3 = \alpha_k^{-1} A'_{21} D_{11}^{+} L_{12}^{\sim} + \alpha^{-1}\alpha_k^{-1} L_{21} \bar{I}_{21} A''_{22} D_{12}^{\sim +} =$$
$$= \alpha_k^{-1} A_{21} W_{11} I_{11} + \alpha^{-1}\alpha_m^{-1}\alpha_k^{-1} \bar{D}_{21} M_{21} A'_{22} W_{12} I_{12}$$
$$\widehat{D} = \alpha(\alpha_r)^{-1} D + (\alpha_r)^{-1} \bar{D},\ \widehat{D}'_{12} = I_{12}^{\lambda^{-1}} \widehat{D}_{12},$$
$$M = \widehat{D}^{-1} L^{-1}, = \widehat{D}^{-1} \begin{pmatrix} L_1^{-1} & 0 \\ -L_4^{-1} L_3 L_1^{-1} & L_4^{-1} \end{pmatrix}$$
$$L_1^{-1} = \widehat{D}'_{12} M_{12} \widehat{D}_{11} M_{11},\ L_4^{-1} = \widehat{D}_{22} M_{22} \widehat{D}_{21} M_{21}$$
$$W = U^{-1}\widehat{D}^{-1} = \begin{pmatrix} U_1^{-1} & -U_1^{-1} U_2 U_4^{-1} \\ 0 & U_4^{-1} \end{pmatrix} \widehat{D}^{-1},$$
$$U_1^{-1} = W_{11} \widehat{D}_{11} W_{21} \widehat{D}_{21},\ U_4^{-1} = W_{12} \widehat{D}'_{12} W_{22} \widehat{D}_{22}.$$

1.4 Зведення всіх формул рекурсивно-блочного алгоритму LDU [1]

(1) Якщо (A = 0) то

$\{\ \alpha_r = \alpha;\quad D = I = J = 0;$
$\quad L = U = \bar{I} = \bar{J} = \bar{D} = I;\quad M = W = \widehat{D} = \alpha I;\ \}$

(2) Якщо (n = 1 & A = [a] & a != 0) то

$\{\ \alpha_r = a;\ L = U = M = W = [a];\ D = [(\alpha * a)^{-1}];\ \widehat{D} = [a^{-2}];$
$\quad J = I = [1];\ \bar{I} = \bar{J} = \bar{D} = [0];\ \}$

(3) Якщо $(n \geq 2\ \&\ A \neq 0)$ то

$\{\ A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$

(3.1)

$$(L_{11}, D_{11}, U_{11}, M_{11}, d_{11}, W_{11}, \alpha_k) = \mathbf{LDU}(A_{11}, \alpha),$$

$$\begin{aligned}
&A_{12}^0 = M_{11} * A_{12};\\
&A_{12}^1 = a_k * \widehat{D}_{11} * A_{12}^0;\\
&A_{12}^2 = \bar{D}_{11} * A_{12}^0/\alpha;\\
&A_{21}^0 = A_{21} * W_{11};\\
&A_{21}^1 = a_k * A_{21}^0 * \widehat{D}_{11};\\
&A_{21}^2 = A_{21}^0 * \bar{D}_{11}/\alpha;
\end{aligned}$$

(3.2)

$$(L_{21}, D_{21}, U_{21}, M_{21}, d_{21}, W_{21}, \alpha_l) = \mathbf{LDU}(A_{21}^2, a_k),$$

(3.3)

$$(L_{12}, D_{12}, U_{12}, M_{12}, d_{12}, W_{12}, \alpha_m) = \mathbf{LDU}(A_{12}^2, a_k),$$

$$\begin{aligned}
&\lambda = \tfrac{a_l}{a_k}; a_s = \lambda * a_m;\\
&A_{22}^0 = A_{21}^1 * D_{11}^+ * A_{12}^1;\\
&A_{22}^1 = (\alpha a_k^2 * A_{22} - A_{22}^0)/(\alpha a_k);\\
&A_{22}^2 = \bar{D}_{21} * M_{21} * A_{22}^1 * W_{12} * \bar{D}_{12};\\
&A_{22}^3 = A_{22}^2/(a_k^2\alpha);
\end{aligned}$$

(3.4)

$$(L_{22}, D_{22}, U_{22}, M_{22}, d_{22}, W_{22}, \alpha_r) = \mathbf{LDU}(A_{22}^3, a_s),$$

$$\begin{aligned}
&J_{12}^\lambda = \lambda * J_{12} + \bar{J}_{12}; \quad I_{12}^\lambda = \lambda I_{12} + \bar{I}_{12};\\
&L_{12}^\sim = L_{12} * I_{12}^\lambda; \quad U_{12}^\sim = J_{12}^\lambda * U_{12};\\
&U_2 = J_{11} * M_{11} * A_{12}/a_k + J_{21} * M_{21} * A_{22}^1/(a_l * \alpha);\\
&L_3 = A_{21} * W_{11} * I_{11}/a_k + \bar{D}_{21} * M_{21} * A_{22}^1 * W_{12} * I_{12}/(a_m * a_k * \alpha);
\end{aligned}$$

$$L = \begin{pmatrix} L_{11}L_{12}^\sim & 0 \\ L_3 & L_{21}L_{22} \end{pmatrix}, \quad D = \begin{pmatrix} D_{11} & \lambda^{-2}D_{12} \\ D_{21} & D_{22} \end{pmatrix}, \quad U = \begin{pmatrix} U_{21}U_{11} & U_2 \\ 0 & U_{22}U_{12}^\sim \end{pmatrix},$$

$$\widehat{D} = \alpha(\alpha_r)^{-1}D + (\alpha_r)^{-1}\bar{D}, \ \widehat{D}'_{12} = I_{12}^{\lambda^{-1}}\widehat{D}_{12},$$

$$L_1^{-1} = \widehat{D}'_{12}M_{12}\widehat{D}_{11}M_{11}, \ L_4^{-1} = \widehat{D}_{22}M_{22}\widehat{D}_{21}M_{21},$$

$$M = \widehat{D}^{-1} \begin{pmatrix} L_1^{-1} & 0 \\ -L_4^{-1} L_3 L_1^{-1} & L_4^{-1} \end{pmatrix},$$

$$U_1^{-1} = W_{11} \widehat{D}_{11} W_{21} \widehat{D}_{21}, \; U_4^{-1} = W_{12} \widehat{D}'_{12} W_{22} \widehat{D}_{22},$$

$$W = \begin{pmatrix} U_1^{-1} & -U_1^{-1} U_2 U_4^{-1} \\ 0 & U_4^{-1} \end{pmatrix} \widehat{D}^{-1}$$

}

Розділ 2: Опис реалізації багатопоточної рекурсивної модернізації за допомогою засобів Java для роботи з потоками.

2.1. Опис засобів Java для роботи з потоками

Багатопоточність у Java є подібним підходом до багатопроцесорності. Однак між ними є деякі принципові відмінності. Замість фізичного процесора багатопоточність включає віртуальні та незалежні потоки. Він призначає кожному процесу один або кілька потоків на основі їх складності. Кожен потік є віртуальним і незалежним від іншого. Це робить виконання процесу набагато безпечнішим. Якщо під час несподіваної ситуації припиняється один або два потоку, виконання процесу не зупиниться.

Життєвий цикл кожного потоку в Java має п'ять різних етапів. Етапи життєвого циклу потоку: New, Runnable, Running, Waiting, Dead.

New - на цьому етапі ініціюється потік. Після цього кожен потік залишається в стані New, доки потоку не буде призначено нове завдання.

Runnable - тут потік призначається завданню і встановлюється для виконання завдання.

Running - тут потік запускається, коли управління входить до потоку, і потік виконує завдання і продовжує виконання, поки не завершить роботу.

Waiting - іноді існує ймовірність того, що один процес в цілому може залежати від іншого. Під час такої зустрічі потік може зупинитися для отримання проміжного результату через його залежність від іншого процесу. Цей етап називається етапом очікування.

Dead - завершальним етапом виконання процесу з багатопоточністю в Java є завершення потоку. Після завершення процесу JVM автоматично оголошує потік мертвим і припиняє його.

За допомогою багатопотокових засобів ми можемо досягти лінійного прискорення програми. Ми використовуємо всі потужності нашої машини, на сьогодні це є дуже важливою складовою, бо майже кожна система має більше ніж один процесор, де в кожному процесорі може бути більше ніж один потік.

Але й в багатопотоковій системі можуть бути свої недоліки. Перше - це те, що набагато складніше писати програми, це тягне за собою проблеми виправлення помилок та читабельності для інших розробників. Також є вірогідність помилок у виконанні програми таких, як:

- Deadlock[5] - може виникнути в ситуації, коли потік очікує блокування об'єкта, яке отримує інший потік, а другий потік чекає блокування об'єкта, яке отримує перший потік. Зовнішньо, але потоки чекають для всіх інших, щоб захистити блокування, умова називається deadlock.
- Livelock[6] - це ще одна проблема паралельності, яка схожа на Deadlock. У Livelock два або більше потоків продовжують передавати стани між собою замість того, щоб чекати нескінченно, як ми бачили в прикладі тупикової блокування. Отже, потоки не можуть виконувати свої відповідні завдання.Чудовим прикладом блокування є система обміну повідомленнями, де, коли виникає виняток, споживач повідомлення відкочує транзакцію і повертає повідомлення в голову черги. Потім одне й те саме повідомлення багаторазово зчитується з черги, лише щоб викликати ще один виняток і

повертатися в чергу. Споживач ніколи не візьме жодне інше повідомлення з черги.

- Starvation[7] - описує ситуацію, коли потік не може отримати регулярний доступ до спільних ресурсів і не може досягти прогресу. Це трапляється, коли спільні ресурси стають недоступними на тривалий період через «жадібні» потоки. Наприклад, припустимо, що об’єкт надає синхронізований метод, повернення якого часто займає багато часу. Якщо один потік часто викликає цей метод, інші потоки, які також потребують частого синхронізованого доступу до того самого об’єкта, часто блокуються.

2.2. Модернізації рекурсивного алгоритму за допомогою засобів Java для роботи з потоками.

2.2.1 Паралельна LDUMW факторизація матриці засобами Java

Для написання паралельного алгоритму за допомогою засобів Java, була надана схема розбиття елементів програми на відповідні рівні. Кожен рівень складається з деякої кількості кроків, наприклад, рівень 3 містить кроки 2, 3, 4 та 5. Кожен крок програми був окремо реалізований за допомогою написання власного класу який імплементує інтерфейс Callable. Callable використовується для виклику задач паралельно з програми з результатом виклику. Кожний рівень у схемі є окремими паралельними кроками які доходять до точки синхронізації і очікують виконання попереднього рівня. Наприклад, кроки 2, 3, 4 та 5 повинні повністю виконатися, щоб почалося виконання кроків 6, 7 та 8.

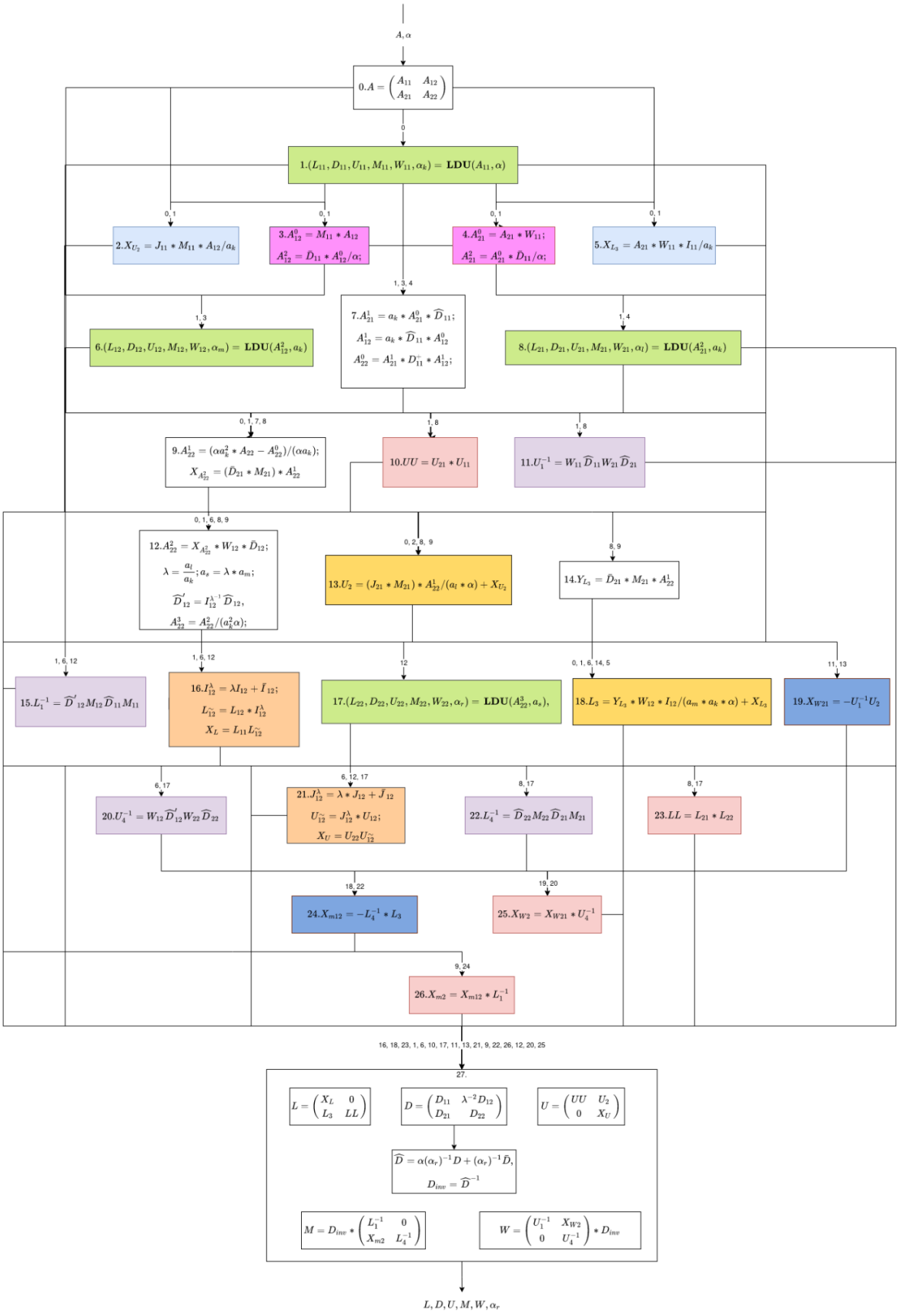


Рис. 1 Схема виконання блочно рекурсивного алгоритму

Вигляд розпаралелення алгоритму за допомогою засобів Java виглядає наступним чином:

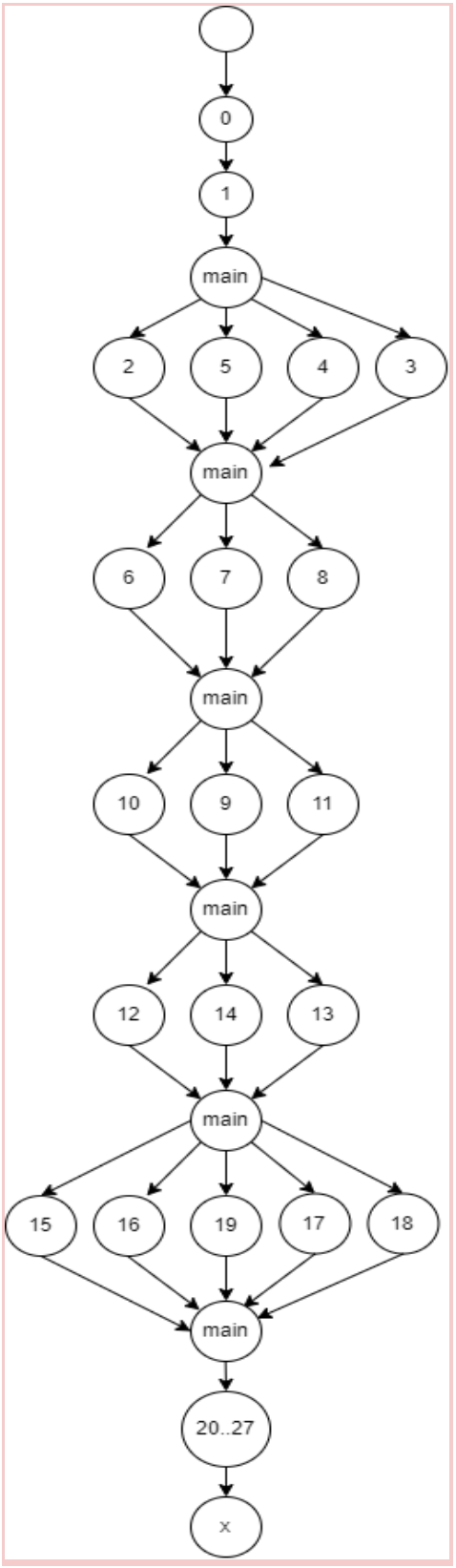


Рис. 2 Розпаралелення LDU факторизації матриці

Для кожного можливого до розпаралелення кроку було створено Callable<MatrixS> реалізація:

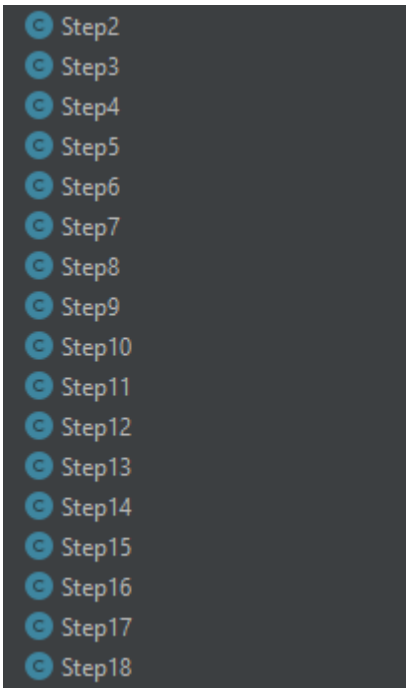


Рис. 3 Класи створені для кожного кроку

Приклад одного з таких класів:

```
public class Step2 implements Callable<MatrixS> {
    private final MatrixS J;
    private final MatrixS M;
    private final MatrixS A12;
    private final Ring ring;

    public Step2(MatrixS j, MatrixS m, MatrixS a12, Ring ring) {
        J = j;
        M = m;
        A12 = a12;
        this.ring = ring;
    }

    @Override
    public MatrixS call() throws Exception {
        return (J.multiplyMT(M, ring)).multiplyMT(A12, ring);
    }
}
```

Рис. 4 Клас другого кроку з тяжкою операцією

Вигляд рівня у точці синхронізації:

```
//--------------------------------------STEP2-----------------------------------------------------
Callable<MatrixS> step2Callable = new Step2(F11.J, F11.M, A12, ring);
MatrixS U2 = null;
//--------------------------------------END STEP2-------------------------------------------------


//--------------------------------------STEP3-----------------------------------------------------
Callable<MatrixS[]> step3Callable = new Step3(F11.M, F11.Dbar, A12, a, ring);
MatrixS A12_0 = null;
MatrixS A12_2 = null;
//--------------------------------------END STEP3-------------------------------------------------


//--------------------------------------STEP4-----------------------------------------------------

Callable<MatrixS[]> step4Callable = new Step4(A21, F11.W, F11.Dbar, ring, a);
MatrixS A21_0 = null;
MatrixS A21_2 = null;
//--------------------------------------END STEP4-------------------------------------------------



Future<MatrixS> futureStep2 = EXECUTOR_SERVICE.submit(step2Callable);
Future<MatrixS[]> futureStep3 = EXECUTOR_SERVICE.submit(step3Callable);
Future<MatrixS[]> futureStep4 = EXECUTOR_SERVICE.submit(step4Callable);

//--------------------------------------STEP5-----------------------------------------------------
MatrixS L3 = (A21.multiplyMT(F11.W.multiplyMT(F11.I, ring), ring));
//--------------------------------------END STEP5-------------------------------------------------
```

Рис. 5 Приклад третього рівня паралельних кроків

Отримання результатів у поточного рівня схеми:

```
try {

    U2 = futureStep2.get(); // STEP2
    MatrixS[] step3Res = futureStep3.get(); // STEP3
    MatrixS[] step4Res = futureStep4.get(); // STEP4

    A12_0 = step3Res[0]; // STEP3
    A12_2 = step3Res[1]; // STEP3

    A21_0 = step4Res[0]; // STEP4
    A21_2 = step4Res[1]; // STEP4
} catch (Exception e) {
    EXECUTOR_SERVICE.shutdown();
    throw e;
}
```

Рис. 6 Приклад отримання результатів з третього рівня схеми

Для запуску кожного кроку паралельно використовується спеціальний сервіс - ForkJoinPool.

ForkJoinPool є реалізацію ExecutorService, яка керує робочими потоками та надає нам інструменти для отримання інформації про стан і продуктивність контейнеру потоків. Робочі потоки можуть одночасно виконувати лише одне завдання, але ForkJoinPool не створює окремий потік для кожної окремої підзадачі. Замість цього кожен потік у пулі має свою власну подвійну чергу (або deque, яка вимовляється), яка зберігає завдання. Ця архітектура життєво важлива для балансування робочого навантаження потоку за допомогою алгоритму роботи.

В моїй реалізації використовується ForkJoinPool для запуску кожного кроку (потоку) програми:

```
public final static ForkJoinPool EXECUTOR_SERVICE = new ForkJoinPool();
```

Рис. 7 Об'явлення та ініціалізація сервіса для запуску потоків

2.2.2 Багатопоточне рекурсивне множення використане в розпаралеленому алгоритмі

Звичайне множення складається з 8 операцій:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 + 2 \cdot 1 & 1 \cdot 0 + 2 \cdot 1 \\ 3 \cdot 0 + 4 \cdot 1 & 3 \cdot 0 + 4 \cdot 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 4 & 4 \end{pmatrix}$$

Рис. 8 Приклад множення та всіх операцій

Було створено клас, який реалізує RecursiveTask. Завдання, які можна розділити на незалежні підзадачі, і кінцевий результат завдання

можна отримати з результатів підзадачі, можна більш ефективно реалізувати за допомогою RecursiveTask.

```
public class RecursiveMultiplication extends RecursiveTask<MatrixS> {
    private final Ring ring;
    private final MatrixS matrixA;
    private final MatrixS matrixB;

    public RecursiveMultiplication(MatrixS matrixA, MatrixS matrixB, Ring ring) {
        this.matrixA = matrixA;
        this.matrixB = matrixB;
        this.ring = ring;
    }
```

Рис. 9 Власний клас, який реалізує RecursiveTask

В методі compute() нашої реалізації прописаний звичайний алгоритм множення матриць.

```
@Override
protected MatrixS compute() {
    if (matrixA.M.length <= THRESHOLD) return matrixA.multiplyRecursiveMTh(matrixB, ring);

    MatrixS[] matrixASplit = matrixA.split();
    MatrixS[] matrixBSplit = matrixB.split();
    RecursiveMultiplication matrix1 = new RecursiveMultiplication(matrixASplit[0], matrixBSplit[0], ring);
    RecursiveMultiplication matrix2 = new RecursiveMultiplication(matrixASplit[1], matrixBSplit[2], ring);
    RecursiveMultiplication matrix3 = new RecursiveMultiplication(matrixASplit[0], matrixBSplit[1], ring);
    RecursiveMultiplication matrix4 = new RecursiveMultiplication(matrixASplit[1], matrixBSplit[3], ring);
    RecursiveMultiplication matrix5 = new RecursiveMultiplication(matrixASplit[2], matrixBSplit[0], ring);
    RecursiveMultiplication matrix6 = new RecursiveMultiplication(matrixASplit[3], matrixBSplit[2], ring);
    RecursiveMultiplication matrix7 = new RecursiveMultiplication(matrixASplit[2], matrixBSplit[1], ring);
    RecursiveMultiplication matrix8 = new RecursiveMultiplication(matrixASplit[3], matrixBSplit[3], ring);

    matrix1.fork();
    matrix2.fork();
    matrix3.fork();
    matrix4.fork();
    matrix5.fork();
    matrix6.fork();
    matrix7.fork();
    matrix8.fork();

    MatrixS[] results = new MatrixS[4];
    results[0] = matrix1.join().addMultiThreads(matrix2.join(), ring);
    results[1] = matrix3.join().addMultiThreads(matrix4.join(), ring);
    results[2] = matrix5.join().addMultiThreads(matrix6.join(), ring);
    results[3] = matrix7.join().addMultiThreads(matrix8.join(), ring);

    return MatrixS.join(results);
}
```


Рис. 10 Метод, який виконується на кожному рекурсивному кроці.

Ця бібліотека ідеально підходить для багатопоточних рекурсивних програм.

Розділ 3: Опис DAP системи, динамічної системи розрахунків, кроків алгоритму, та основним складовим цієї системи.

В нас є деяка машина на якій ми хочемо розпаралелити процес вирахування рекурсивно-блочного алгоритму за допомогою деякої кількості процесорів. За допомогою наданого графу, можна легко імплементувати та виконати обчислення, цей граф надає розбиття рекурсивно блочного алгоритму на дропи, кожен з яких має тільки одну складну операцію. Як тільки виконується дроп, його результат відправляється наступному дропу на вхід. Після того як всі дропи виконалися, вони надходять у outputFunction в якій збирається результат до одного цілого. У процесі виконання операцій, процесори оптимально використовуються для обчислення не зв’язних дропів паралельно. Є три етапи виконання:

- надсилання всіх дропів на обчислення за допомогою багатьох потоків
- обчислення дропів
- збір усіх результатів у одне ціле

3.1 Загальний опис основних дій для використання системи DAP

Для того щоб реалізувати свій алгоритм в системі треба:

- створити власний класс, який імплементує клас Drop;
- оголосити арки, які регулюють основний потік вхідних та вихідних даних між дропами;
- перевизначити метод doAmin, в якому до кожної унікальної операції множення треба визначити особистий ключ, а також у правильному порядку додати дропи;

- перевизначити конструктор в якому треба обов’язково визначити арки, тип, кількість вхідних змінних, кількість вихідних даних, а також кількість змінних, які прямують у останню функцію;
- перевизначити метод для послідовного обрахунку, тобто коли ми досягли листової вершини;
- перевизначити метод inputFunction, який обробляє вхідні дані та відправляю їх на обчислення всім дропам;
- перевизначити метод outputFunction, який збирає усі дані з дропів і вираховує фінальний результат;
- перевизначити метод isItLeaf та setLeafSize, які перевіряють чи цей крок є листовою вершиною та виставляє вершину відповідно;

Наступними кроками є конфігурування множення, воно може бути різним, так як множення несиметричне, це ускладнює нам ситуацію.
Для кожного окремого ключа, який ми визначили в попередньому кроці, треба написати кейс в класі MatrSMult4.

Для тестування треба визначити окремий клас, який розширяє DAPTest. В ньоме треба перевизначити:

- метод initData, який визначає початкову матрицю;
- метод checkResult, який перевіряє правильність виконання рохрахунку;
- та метод sequentialExecute, який використовується як стандарт, для порівняння очікуємого з отриманим результатом.

Розділ 4: Реалізація алгоритму на системі DAP

4.1 Використання спеціального об’єкта для зберігання інформації

Було створено спеціальний DTO (Data Transfer Object) об’єкт для зберігання всіх частин LDUMW, які в послідовному алгоритмі зберігалися в тому ж класі. Цей LdumwDto повинен наслідувати Element, щоб його можна було передавати між дропами і використовувати в DAP. LdumwDto складає L, D, U, M, W, Dhat, Dbar, I, Ibar, J, Jbar та a_n. Обчислення цих елементів можна побачити в **1.4. Зведення всіх формул рекурсивно-блочного алгоритму LDU.**

```
public class LdumwDto extends Element {
    private final MatrixS L;
    private MatrixS D;
    private MatrixS Dhat;
    private MatrixS Dbar;
    private final MatrixS U;
    private MatrixS M;
    private MatrixS W;
    private MatrixS I;
    private MatrixS Ibar;
    private MatrixS J;
    private MatrixS Jbar;
    private final Element a_n;
```

Рис. 11 DTO для зберігання всіх матриць рокладу та додаткових елементів

4.2 Основний дроп для LDU факторизації - LdumwFact

4.2.1 Визначення арків

Арки були визначені за допомогою наданої схеми розпаралелення алгоритму LDU факторизації на системі DAP.

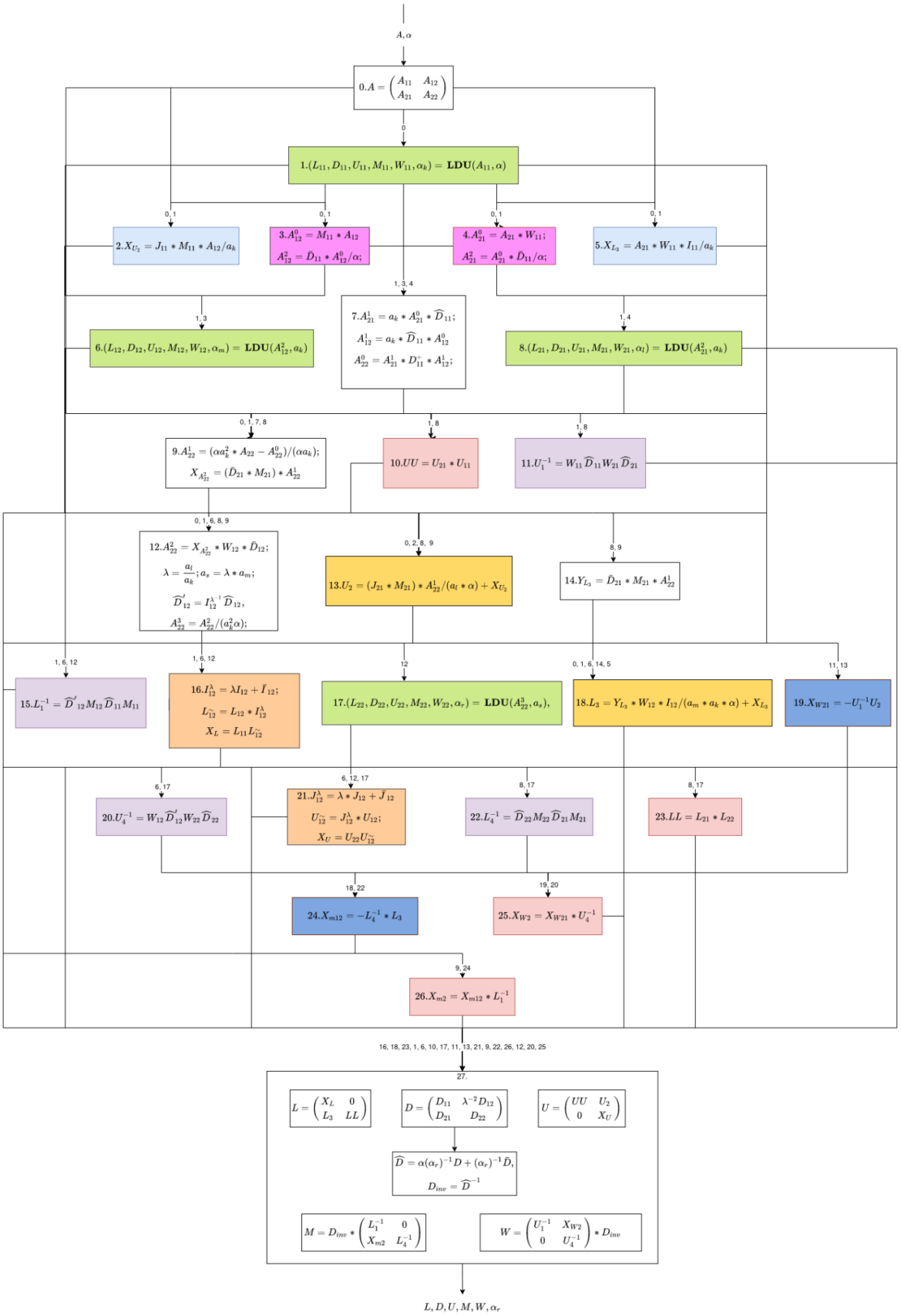


Рис. 12 Схема розподілення обчислень на дропи множення

```
static int[][] arcs_ = new int[][]{
{1, 0, 0, 1, 4, 1, 2, 1, 1, 3, 1, 1, 3, 4, 2, 4, 2, 1, 4, 4, 2, 5, 2, 1, 9, 3, 1, 9, 4, 4, 11, 4, 4, 12, 4, 3, 16, 4, 4},//0. inputFuncti
{2, 0, 0, 3, 0, 0, 4, 0, 0, 5, 0, 0, 6, 1, 1, 7, 0, 0, 8, 1, 1, 9, 0, 0, 10, 0, 0, 11, 0, 3, 14, 0, 0, 16, 0, 2, 19, 0, 0},//1. F11 = LDU
{12, 0, 2},//2. X_U2
{6, 1, 0, 7, 0, 2},//3. A12_0  and  A12_2
{7, 0, 1, 8, 1, 0},//4. A21_0  and  A21_2
{16, 0, 3},//5. X_L3
{11, 0, 1, 14, 0, 1, 16, 0, 1, 17, 0, 1, 19, 0, 2}, //6. F12(am)
{9, 0, 2}, //7. A22_0
{9, 0, 3, 10, 0, 1, 11, 0, 2, 12, 0, 0, 13, 0, 0, 18, 0, 0, 19, 0, 1}, //8. F21(al)
{11, 1, 0, 12, 0, 1, 13, 0, 1}, //9. A22_1  and  X_A22_2
{19, 0, 5}, //10. UU
{14, 0, 2, 15, 1, 1, 15, 2, 0, 17, 0, 2, 19, 0, 4, 19, 3, 8, 19, 1, 9}, //11. lambda  and  as  and  A22_3  and  invD12hat
{19, 0, 6}, //12. U2
{16, 0, 0}, //13. Y_L3 (L3H2)
{19, 0, 12}, //14. X_L
{17, 0, 0, 18, 0, 1, 19, 0, 3}, //15. F22
{19, 0, 11}, //16. L3
{19, 0, 7}, //17. X_U
{19, 0, 10}, //18. LL
{}};//19. OutputFunction
```

Рис. 13 Арки визначенні за наданої схеми

Арки - це масив масивів, кожен з яких представляє один дроп. Цей масив складається з чисел, які розбиті на трійки. Перше число з трійки - це в який дроп буде надсилатися результат даного дропа. Друге число - номер елемента який буде надсилатися з даного дропу. Третє число - номер елемента в який буде надсилатися результат даного елементу дропу.

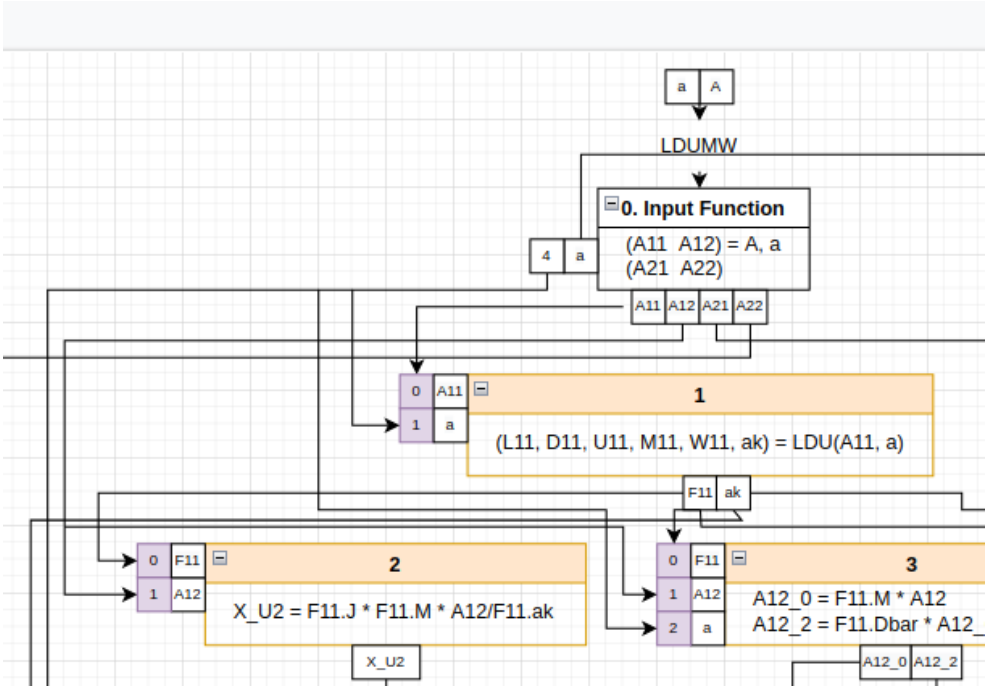


Рис. 14 Приклад дропів та їх елементів з номерами

Наприклад, беремо першу трійку нульового дропу - 1, 0, 0. Перша одиниця представляє дроп в який ми направляємо результат. Цей результат береться з нульового індексу, тобто A11, і він йде в нульовий елемент першого дропу.

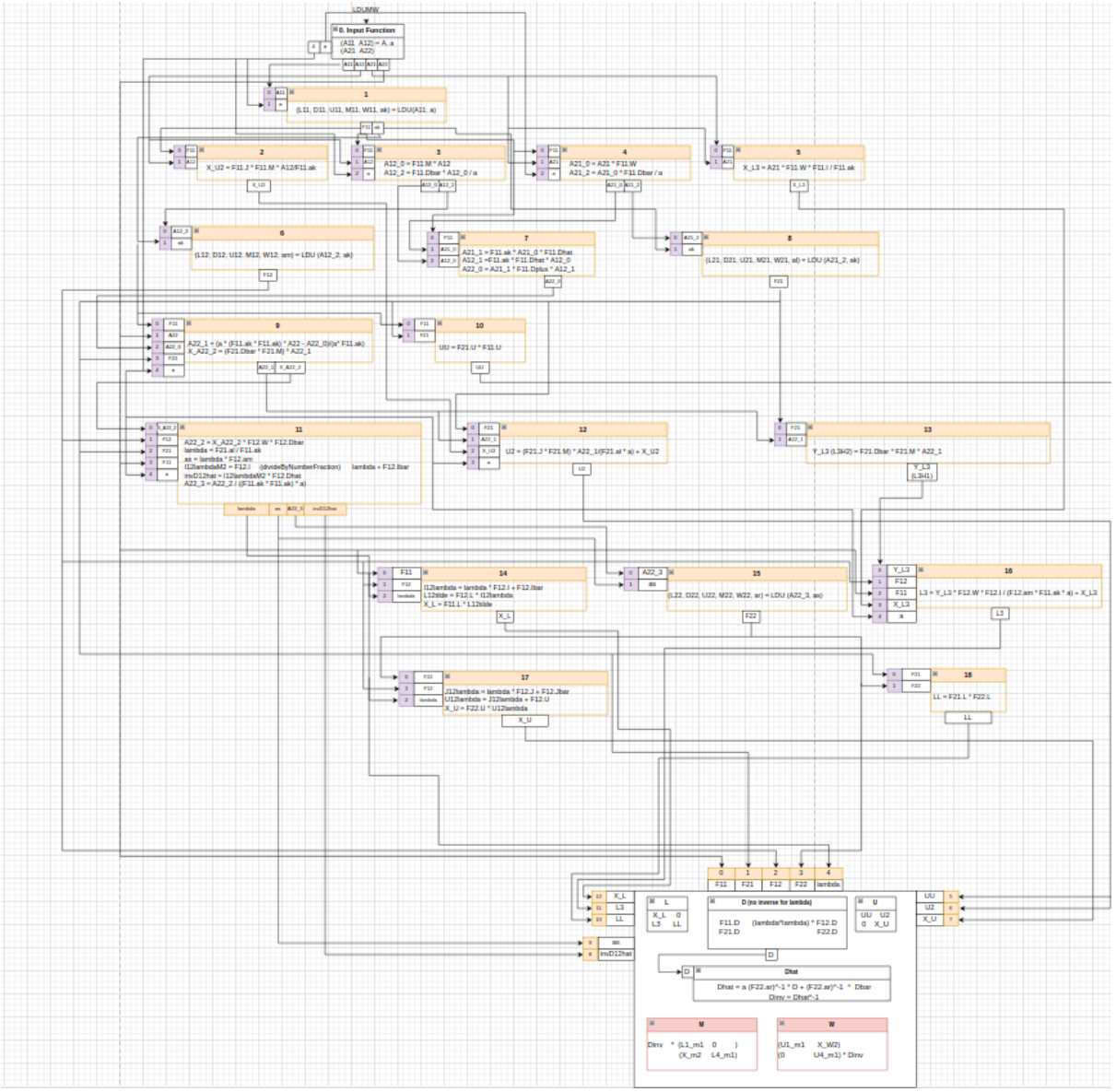


Рис. 15 Повна схема елементів та кожного індексу

4.2.1 Визначення конструктора

В конструкторі ми визначаємо inputDataLength - 2, бо кількість елементів на вхід - 2, це A та a. На вихід - outputDataLength також 2, бо це наше LdumwDto та a_n. resultForOutFunctionLength - 13, бо кількість елементів для генерації результату потребує 13 елементів з дропів.

```
public LdumwFact() {
    arcs = arcs_;
    type = 23;
    number = cnum++;
    inputDataLength = 2;
    outputDataLength = 2;
    inData = new Element[inputDataLength];
    outData = new Element[outputDataLength];
    resultForOutFunctionLength = 13;
}
```

Рис. 16 Конструктор основного дропу LDUMW

4.2.2 Визначення методу doAmin

Наступний метод, який був перевизначений - doAmin.
В ньому визначився порядок дропів за їх номерами. Для множення було визначено окремий ключ, за яким працює switch case, для унікальних складних множень матриць. Наприклад, для кроку 2 визначено ключ 102, який оброблється в класі MatrSMult4.

Порядок додавання дропів в список в цьому випадку є дуже важливим.

```java
@Override
public ArrayList<Drop> doAmin() {
    ArrayList<Drop> amin = new ArrayList<>();
    // step 1
    amin.add(new LdumwFact());
    // step 2
    amin.add(new MatrSMult4());
    amin.get(1).key = 102;
    // step 3
    amin.add(new MatrSMult4());
    amin.get(2).key = 103;
    // step 4
    amin.add(new MatrSMult4());
    amin.get(3).key = 104;
    // step 5
    amin.add(new MatrSMult4());
    amin.get(4).key = 105;
    // step 6
    amin.add(new LdumwFact());
    // step 7
    amin.add(new MatrSMult4());
    amin.get(6).key = 107;
    // step 8
    amin.add(new LdumwFact());
```

Рис. 17 Метод doAmin для визначення ключів та індексів

4.2.3 Визначення методу sequentialCalc

Цей метод перевизначений для випадку, коли дроп є листовою вершиною і ми використовуємо стандартний алгоритм розрахунку LDUMW.

```java
@Override
public void sequentialCalc(Ring ring) {
    MatrixS A = (MatrixS) inData[0];

    Element a = inData[1];
    LdumwDto FF = LDUMW.LDUMWNoInversion(A, a, ring);

    outData[0] = FF;
    outData[1] = FF.A_n();
}
```

Рис. 18 Метод sequentialCalc

4.2.4 Визначення методу inputFunction

В даному методі визначається і записується в спеціальну змінну вхідні дані.

```java
public Element[] inputFunction(Element[] input, Amin amin, Ring ring) {
    MatrixS A = (MatrixS) input[0];
    Element a = input[1];
    MatrixS[] split = A.split();
    Element[] res = new Element[5];
    System.arraycopy(split, i: 0, res, i1: 0, i2: 4);

    res[4] = a;
    return res;
}
```

Рис. 19 Метод inputFunction

4.2.5 Визначення методу outputFunction

Метод використовується для обрахунку фінальних матриць L, D, U, M та W з проміжних результатів дропів.

```
MatrixS UU = (MatrixS) input[5];
MatrixS U2 = (MatrixS) input[6];
MatrixS X_U = (MatrixS) input[7];
MatrixS invD12hat = (MatrixS) input[8];
Element as = input[9];

MatrixS LL = (MatrixS) input[10];
MatrixS L3 = (MatrixS) input[11];
MatrixS X_L = (MatrixS) input[12];

Element lambda2 = lambda.multiply(lambda, ring);
L = MatrixS.join(new MatrixS[]{X_L, MatrixS.zeroMatrix(), L3, LL});
D = MatrixS.join(new MatrixS[]{
        F11.D(), F12.D().multiplyByNumber(lambda2, ring),
        F21.D(), F22.D()});

U = MatrixS.join(new MatrixS[]{UU, U2, MatrixS.zeroMatrix(), X_U});


ldumw = new LdumwDto(L, D, U, F22.A_n());
ldumw.IJMap(a, ring);
```

Рис. 20 Метод outputFunction

4.3 Дроп для множення LDU факторизації

В цьому класі було додано логіку обчислення за унікальними ключами. Приклад доданого коду для ключа 112.

Перший метод - setVars для перевизначення розмірів масивів які будуть передаватися від inputFunction до outputFunction.

```
case (112): {
    inputDataLength = 5;
    outputDataLength = 4;
    resultForOutFunctionLength = 8;
    break;
}
```

Рис. 21 Метод setVars

Наступний метод використовується для підрахунку листових вершин - sequentialCalc. Тут імплементована звичайна послідовна логіка.

```
case (112): {
    MatrixS X_A22_2 = (MatrixS) inData[0];
    LdumwDto F12 = ((LdumwDto) inData[1]);
    LdumwDto F21 = ((LdumwDto) inData[2]);
    LdumwDto F11 = ((LdumwDto) inData[3]);
    Element a = inData[4];

    Element al = F21.A_n();
    Element am = F12.A_n();
    Element ak = F11.A_n();
    MatrixS A22_2 = X_A22_2.multiply(F12.W()
            .multiply(F12.Dbar(), ring), ring);
    Element lambda = al.divideToFraction(ak, ring);
    Element as = lambda.multiply(am, ring);
    Element ak2 = ak.multiply(ak, ring);

    MatrixS I12lambdaM2 = (F12.I()
            .divideByNumbertoFraction(lambda, ring))
            .add(F12.Ibar(), ring);
    MatrixS invD12hat = I12lambdaM2.multiply(F12.Dhat(), ring);
    MatrixS A22_3 = A22_2.divideByNumber(ak2, ring).divideByNumber(a, ring)

    outData[0] = lambda;
    outData[1] = as;
    outData[2] = A22_3;
    outData[3] = invD12hat;
```

Рис. 22 Метод sequentialCalc

Метод inputFunction розбиває вхідні дані на блоки, в нашому випадку це елементи з першого та нульового індексу масиву.

```
case (112): {
    LdumwDto F12 = (LdumwDto) input[1];
      LOGGER.info("-----------------------112 input------------------------------");
    ms = (MatrixS) input[0];
    ms1 = F12.W();
    break;
}
```

Рис. 23 Метод inputFunction

Метод independentCalc використовується для обчислення нескладних операцій, які можна відділити від основних операцій.

```
case (112): {
    LdumwDto F12 = ((LdumwDto) inData[1]);
    LdumwDto F21 = ((LdumwDto) inData[2]);
    LdumwDto F11 = ((LdumwDto) inData[3]);

      LOGGER.info("------------------------112 indepen------------------------------");

    Element al = F21.A_n();
    Element am = F12.A_n();
    Element ak = F11.A_n();

    Element lambda = al.divideToFraction(ak, ring);
    Element as = lambda.multiply(am, ring);
    Element ak2 = ak.multiply(ak, ring);

    MatrixS I12lambdaM2 = (F12.I()
            .divideByNumbertoFraction(lambda, ring))
            .add(F12.Ibar(), ring);
    MatrixS invD12hat = I12lambdaM2.multiply(F12.Dhat(), ring);

    amin.resultForOutFunction[4] = lambda;
    amin.resultForOutFunction[5] = as;
```

Рис. 24 Метод independentCalc

Метод outputFunction в MatrSMult4 використаний для обрахунку фінального результату множення - inputFunction та незалежного результату - independentCalc разом.

```
case (112): {
      LOGGER.info("------------------------112 output------------------------------");
    LdumwDto F12 = (LdumwDto) inData[1];
    Element a = inData[4];
    MatrixS A22_2 = MatrixS.join(resmat).multiply(F12.Dbar(), ring);
    MatrixS A22_3 = A22_2.divideByNumber(input[7], ring).divideByNumber(a, ring);

    res = new Element[]{
            input[4], input[5], A22_3, input[6]
    };
    break;
}
```

Рис. 25 Метод outputFunction

4.4 Клас для тестування дропу LDUMW факторизації

Метод initData використовується для ініціалізації матриці, яка приходить на вхід до LDUMW факторизації.

```
@Override
protected Element[] initData(int size, int density, int maxBits, Ring ring) {
    MatrixS A = matrix(size, density,  maxBits: 5, ring);
    return new Element[]{A, ring.numberONE};
}
```

Рис. 26 Метод initData

Метод перевірки перевірки результату - *checkResult,* використаний для порівняння паралельного алгоритму з послідовним, для отримання і знаходження правильного результату.

```
@Override
protected Pair<Boolean, Element> checkResult(DispThread dispThread, String[] args, Element[] initData, Element[] resultData, Ring ring)
    LdumwDto ldumwDto = (LdumwDto) resultData[0];
    ldumwDto.setD(LdumwFact.invForD(ldumwDto.D(), ring));

    MatrixS A = (MatrixS) initData[0];
    Element a = initData[1];

    LdumwDto ldumwDtoSequential = LDUMW.LDUMW(A, a, ring);

    if (ldumwDto.equals(ldumwDtoSequential)) {
        return new Pair<>(true, null);
    }

    return new Pair<>(false, null);
}
```

Рис. 27 Метод checkResult

І останній метод цього класу - sequentialExecute, котрий викликає послідовний алгоритм LDUMW факторизації.

```
@Override
protected Element[] sequentialExecute(Element[] data, Ring ring) {
    MatrixS A = (MatrixS) data[0];
    Element a = data[1];

    LdumwDto FF = LDUMW.LDUMW(A, a, ring);

    return new Element[] {FF, FF.A_n()};
}
```

Рис. 28 Метод sequentialExecute

Розділ 5: Проведення експериментів розроблених алгоритмів на системі DAP та за допомогою засобів Java.

Головним завданням експериментів було знаходження оптимальної величини листової вершини. Для цього було подано на вхід матриця з щільністю 100 в якій були використані цілі числа. Розмір матриці також довільний, але він повинен бути ступенем двійки, тобто 2, 4, 8, 16 ….

В таблицях використовуються такі стовпчики як розмір матриці, однопоточний варіант, багатопоточний варіант, прискорення та кількість спроб.

Однопоточний варіант - це час обчислення матриці за допомогою звичайного послідовного алгоритму.

Багатопоточний алгоритм - це алгоритм, який був використаний у даному порівнянні.

Прискорення - це відношення часу виконання послідовного алгоритму до багатопоточного варіанту.

Кількість спроб - кількість виконаних ітерацій однієї операції на ділення всього часу на кількість спроб.

Специфікація машини на якій проведені експерименти:

Процесор: Intel Core i5-9300HF (2.4 - 4.1 ГГц)

Кількість ядер: 4

Кількість потоків: 8

Оперативна пам’ять: 16гб (2400 МГц)

5.1 Перевірка правильності результатів

Для того щоб перевірити, що результати правильні, то треба підставити результуючі матриці в рівняння нижче[1].

$A = LDU,\ L\widehat{D}M = \mathbf{I},\ W\widehat{D}U = \mathbf{I}$:

Така перевірка зроблена на розробленому алгоритмі, що доводить правильність функціоналу алгоритму.

5.2 Проведення експериментів на багатопотоковій модернізації розроблень засобами Java

Тести були проведені на послідовному алгоритмі та алгоритмі описаному в пункті **2.2.1 Паралельна LDUMW факторизація матриці засобами Java.** Всі запуски програми були зроблені на **4 процесорах.**

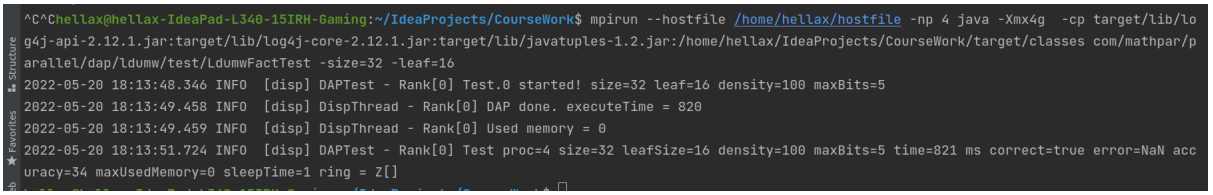
Матриця генерується з такими параметрами:

- довільний розмір, який є результатом піднесення до степеня двійки
- щільність матриця 100 од.
- кількість біт - 5
- тип даних - цілі числа Z[]

Команда запуску з розміром матриці - 256 та листовою вершиною - 16:

mpirun --hostfile /home/user/hostfile -np 4 java -Xmx4g -cp target/lib/log4j-api-2.12.1.jar:target/lib/log4j-core-2.12.1.jar:target/lib/javatuples-1.2.jar:/home/user/../target/classes com/mathpar/parallel/dap/ldumw/test/LdumwFactTest -size=256 -leaf=16

Результат виконання програми:



Виконується правильно, флаг correct = true.

Розмір матриці /	Однопоточни	Багатопоточний	Прискорення	Кількість спроб

Час виконання (мілісекунди)	й варіант	варінт		(задля усереднення)
16	797	799	0%	100
32	10768	9797	9.9%	100
64	81699	68425	19%	40
128	898683	645725	39%	20
256	1339338	986053	35%	1
512	20163450	13710580	47%	1

Як ми можемо бачити, чим більше матриця стає за розміром, тим кращі результати ми отримали. Прискорення значне відбулося завдяки рекурсивному паралельному множенні.

Результати без багатопоточного множення ми можемо бачити в наступній таблиці:

Розмір матриці / Час виконання (мілісекунди)	Однопоточни й варіант	Багатопоточний варінт	Прискорення	Кількість спроб (задля усереднення)
16	797	799	0%	100
32	10768	10609	1.4%	100
64	81699	81115	0,7%	40
128	898683	861574	4.3%	20
256	1339338	1281191	4.5%	1
512	20163450	19070381	5.7%	1

Алгоритм без багатопотокового множення має такуж характеристику, тобто чим більше матриця, тим більше прискорення. Але різниця надзвичайна.

5.3 Проведення експериментів на багатопотоковій модернізації розроблений на системі DAP

Головним завданням даного розділу було знаходження оптимальної листової вершини.

Далі будуть надані результати експериментів з різними листовими вершинами (2, 4, 8, 16, 32, 64 та 128) та розмірами матриць від 16x16 до 256x256.

Листова вершина - 2:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	2	4	797	130	6.13
32	2	4	10768	669	16,09
64	2	4	81699	5932	13.7
128	2	4	898683	96335	9.32
256	2	4	1339338	338623	0.57

Листова вершина - 4:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	4	4	797	328	2.42
32	4	4	10768	1006	10.70
64	4	4	81699	6463	12.64
128	4	4	898683	80767	11.12
256	4	4	1339338	830389	0.62

Листова вершина - 8:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	8	4	797	167	4.77

32	8	4	10768	1107	9.72
64	8	4	81699	5617	14.54
128	8	4	898683	74970	11.98
256	8	4	1339338	937536	0.70

Листова вершина - 16:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	16	4	797	108	7.37
32	16	4	10768	769	14.00
64	16	4	81699	5835	14.00
128	16	4	898683	77831	11.54
256	16	4	1339338	1887373	0.71

Листова вершина - 32:

Розмір матриці /	Розмір листової	Кількість процесорі	Однопоточний	Багатоп оточний	Прискорення

Час виконання (мілісекунди)	вершини	в	варіант(мс)	варінт(мс)	
16	32	4	797	114	6.99
32	32	4	10768	543	19.83
64	32	4	81699	5152	15.85
128	32	4	898683	80816	11.12
256	32	4	1339338	1699210	0.78

Листова вершина - 64:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	64	4	797	97	8.21
32	64	4	10768	516	20.86
64	64	4	81699	5436	15.029
128	64	4	898683	89968	9.98
256	64	4	1339338	1854117	0.72

Листова вершина - 128:

Розмір матриці / Час виконання (мілісекунди)	Розмір листової вершини	Кількість процесорів	Однопоточний варіант(мс)	Багатопоточний варінт(мс)	Прискорення
16	128	4	797	97	8.21
32	128	4	10768	516	20.86
64	128	4	81699	5436	15.029
128	128	4	898683	85397	10.52
256	128	4	1339338	2244306	0.59

Підсумовуючи результати досліджень можемо бачити, що на даній машині при розмірі 256 на 256 елементів прискорення є меншим за одиницю, тобто однопоточний алгоритм виконується швидше за багатопоточний. Але при розмірі листової вершини 32 отримуємо найкращі результати на матрицях до розміру 256 на 256. Можливо це специфіка даної машини, при більш потужному процесорі, результати очікуються кращі на розмірі 256 на 256.

Графік прискорення з відношенням до розміру матриці з листовою вершиною 32.

Висновки

Були розроблені багатопотокові алгоритми LDUMW факторизації за допомогою стандартних засобів багатопоточності Java та на системі DAP. Також була досліджена система DAP та всі її тонкощі.

Після виконаних досліджень на системі DAP було виявлено, що коли обчислення доходять до матриці великих розмірів, для даної машини - це матриця 256 на 256 елементів, відбувається значне уповільнення. Але на величинах 64, 128 прискорення значне - 10 - 20 разів.

Паралельний алгоритм розроблений за допомогою стандартної бібліотеки для багатопоточності Java не відрізняється такою особливістю. Тобто не має сповільнення на розмірі 256. Загальне прискорення досягає 1.5 - 2 раза на розмірах до 256x256 елементів.

Проведені експерименти на різних листових вершинах показали, що на даній машині з 4 процесорами листова вершина розміром 32 є найоптимальнішою.

Код розроблений на основі репозиторія dap01:
https://bitbucket.org/mathpar/dap01/src/master/

Список використаних джерел

1. Malaschonok G. I. LDU factorization [Електронний ресурс] / Gennadi I. Malaschonok. – 2020. – Режим доступу до ресурсу: https://arxiv.org/abs/2011.04108.
2. LU Factorization [Електронний ресурс] // Studies in Computational Mathematics, 2006. – 2006. – Режим доступу до ресурсу: https://www.sciencedirect.com/topics/mathematics/lu-factorization.
3. A = LU and Solving Systems [Електронний ресурс] – Режим доступу до ресурсу: https://staff.imsa.edu/~fogel/LinAlg/PDF/14%20Solving%20Factored%20Systems.pdf.
4. Learn Multithreading in Java With Examples [Електронний ресурс] // Simplilear. – 2022. – Режим доступу до ресурсу: https://www.simplilearn.com/tutorials/java-tutorial/multithreading-in-java.
5. Deadlock in Java [Електронний ресурс] // javatpoint – Режим доступу до ресурсу: https://www.javatpoint.com/deadlock-in-java.
6. Java Thread Deadlock and Livelock [Електронний ресурс] // Kamlesh Kumar. – 2021. – Режим доступу до ресурсу: https://www.baeldung.com/java-deadlock-livelock.

7. Starvation and Livelock [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html.
8. Malaschonok, G.I., Sidko, A.A. Supercomputer Environment for Recursive Matrix Algorithms. *Program Comput Soft* 48, 90–101 (2022). https://doi.org/10.1134/S0361768822020086
9. *Малашонок Г.І., Сідько А.А.* Розподілені обчислення: ДАП - технологія розпаралелювання рекурсивних алгоритмів. Наукові записки НаУКМА. Комп'ютерні науки. 2018. Том 1. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/144796
10. *Malaschonok G. I., Sidko A. A.* Parallel computer algebra: a new scheme for controlling the parallelization of matrix recur-sive algorithms. Fifth International Conference on High Perfor-mance Computing (HPCUA 2018) being held October 22-23, 2018 in Kyiv, Ukraine.2018. P. 77–85. [Електронний ресурс]. – режим доступу: http://hpc-ua.org/hpc-ua-18/files/proceedings/13.pdf
11. *Malaschonok G., Sidko A.* Control of matrix computations on distributed memory. International conference Polynomial Computer Algebra. St. Petersburg, PDMI RAS, 2019. P. 94–99. [Електронний ресурс]. – режим доступу: https://pca-pdmi.ru/2019/files/21/pca_MalaschSidko.pdf.