Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ”
Кафедра інформатики факультету інформатики

АЛГОРИТМ SVD ДЛЯ РОЗПОДІЛЕНОЇ ПАМ’ЯТІ

Текстова частина

магістерської роботи

за спеціальністю “Інженерія програмного забезпечення” 121

Керівник магістерської роботи
д.ф.-м.н., проф. Малашонок Г.І.

_____

11 червня 2020 р.

Виконав студент 2 курсу
Сірош І.А.
11 червня 2020 р

Київ 2020

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
к.ф-м.н.
_____ С. С. Гороховський
(підпис)
7 листопада 2019 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на дипломну роботу

студенту 2 року МП ІПЗ факультету інформатики
Сірошу Іллі Андрійовичу
Розробити Паралельний алгоритм для обчислення SVD
Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1 SVD на основі ітераційного алгоритму QR-розкладу
2 Використання методу матриць Гівенса для паралельного обчислення ортоганального розкладу
2.1 Алгоритм симетричного двостороннього зведення до тридіагональної матриці на n^2 процесорах
2.2 Алгоритм симетричного двостороннього зведення до тридіагональної матриці на n процесорах
3 Результати тестування алгоритмів
Висновки
Список літератури
Додатки

Дата видачі 25 жовтня 2019 р.

Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Алгоритм SVD для розподіленої пам’яті

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проєкту (роботи)	Термін виконання етапу	Примітка
1	Затвердження теми магістерської роботи	21.10.2019	
2	Огляд матеріалів за темою	15.11.2019	
3	Реалізація практичної частини першого розділу	31.12.2019	
4	Проведення тестування на кластері QR-розкладу	16.03.2020	
5	Реалізація практичної частини другого розділу	17.02.2020	
6	Проведення тестування на кластері UTV-розкладу	20.04.2020	
7	Оформлення результатів тестування	04.05.2020	
8	Оформлення текстової частипи та підготовка презентації	27.05.2020	
9	Корегування роботи за результатами попереднього захисту	05.06.2020	
10	Захист магістерської роботи	16.06.2020	

Студент Сірош І.А.

Керівник Малашонок Г.І.

9 червня 2020 р

ЗМІСТ

Анотація

В роботі розглянуто блочно-рекурсивний алгоритм QR-розкладу та алгоритм UTV-розкладу на основі матриць Ґівенса. Розроблено та реалізовано їх паралельні варіанти. Проведено тестування та зроблено висновки щодо їх подальшого вдосконалення.

Ключові слова: *SVD, QR-розклад, UTV-розклад, матриці Ґівенса, паралельні обчислення.*

ВСТУП

На сьогоднішній день SVD (Singular Value Decomposition — сингулярний розклад матриці) має велику кількість застосувань, тож виникає потреба у пришвидшені виконання обчислення. Так постає проблема паралельного обчислення SVD. Стандартний алгоритм SVD розділяють на два етапи: (1) бідіагоналізація вхідної матриці та (2) ітеративне обчислення SVD бідіагональної матриці. Зазвичай перший етап це ітеративне приведення вхідної матриці за допомогою обчислення QR-розкладу до бідіагональної матриці.

В цій роботі зосередимось на першому етапі обчислення SVD. Розглянемо можливості пришвидчити його виконання за допомогою паралелізації. Реалізуємо різні варіанти обчислення першого етапу, протестуваємо їх та проаналізуємо прискорення.

Робота складається із трьох розділів.

У першому розділі запропонуємо реалізацію паралельного блочно-рекурсивного алгоритму QR-розкладу на основі матирць Ґівенса.

У другому розділі розглянемо реалізації паралельного алгоритму UTV-розкладу на основі матриць Ґівенса.

У третьому розділі наведемо результати тестувань реалізованих параельних алгоритмів із першого та другого розділів, та проаланізуємо прискорення порівняно із послідовним обчисленням.

1 Ітеративне обчислення SVD за допомогою обчислення QR-розкладу

1.1 Попередня інформація

QR-розклад це факторизація матриці A на добуток ортоганальної матриці Q та верхної трикутної матриці R, тобто $A = QR$. Також справедлива рівність $Q^T A = R$, адже $Q^{-1} = Q^T$. За замовченням, ми розглядаємо алгоритм розкладу для квадратної матриці $A_{2^n \times 2^n}$ над полем дійсних чисел.

Розглянемо частковий випадок для матриці 2×2. Тоді розклад $A = QR$ буде мати наступну форму:

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a & b \\ 0 & d \end{pmatrix}, \text{ або відповідно } \quad \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a & b \\ 0 & d \end{pmatrix}$$

де числа s та c задовольняють нерівність $s^2 + c^2 = 1$.

Якщо $\gamma = 0$, тоді $c = 1, s = 0$. Якщо $\gamma \neq 0$, тоді $c = \frac{\gamma}{\Delta}, s = \frac{\alpha}{\Delta}$, де $\Delta = \sqrt{\alpha^2 + \gamma^2}$. Матрицю $Q_{2\times 2}$ надалі будемо позначати як $g_{\alpha,\gamma}$.

1.1.1 Матриці Ґівенса

Нехай маємо матрицю A, (i, j) та $(i+1, j)$ елементами якої є α та γ, і всі елементи зліва від них є нульовими: $\forall (s < j) : (a_{i,s} = 0)\ \&\ (a_{i+1,s} = 0)$. Можна

$$G_{i,j} = \mathrm{diag}(I_{i-1}, g_{\alpha,\gamma}, I_{n-i-1}) = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c & s & & \\ & & & -s & c & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}$$

визначити матрицю Ґівенса:

Тоді результат добутку $G_{i,j}A$ відрізняється від A тільки двома рядками i та $i+1$, водночас всі елементи зліва від стовпчика j залишаються нульовими, а елемент

$a_{i+1,j}$ стає нульовим. Ця властивість матриці Ґівенса дозволяє сформулювати наступний алгоритм.

1.1.2 Матриці Ґівенса для QR-розкладу

(1) Почнемо із лівого сповпця і обнулимо в ньому всі елемент, що розташовані під діагоналлю:

$$A_1 = G_{1,1}G_{2,1}\ldots G_{n-2,1}G_{n-1,1}A \tag{1.1}$$

(2) Далі обнулимо елементи під діагоналлю у другому сповпчику:

$$A_2 = G_{2,2}G_{3,2}\ldots G_{n-2,2}G_{n-1,2}A_1 \tag{1.2}$$

(k) Визначимо $G_{(k)} = G_{k,k}G_{k-1,k}\ldots G_{n-2,k}G_{n-1,k}$, для $k = 1,2,\ldots,n-1$. Тоді для обнулення елементів k-го сповпчика треба обчислити наступний добуток:

$$A_k = G_{(k)}A_{k-1}$$

(n-1) Вкінці обчислення буде обнулено один елемент в стопвчику $n-1$:

$$A_{n-1} = G_{(n-1)}A_{n-2} = G_{n-1,n-1}A_{n-2}$$

Оцінимо кількість операцій в алгоритмі. Необхідно обчислити $(n^2-n)/2$ матриць Ґівенса та виконати 6 операцій (2 піднесення до квадрату, 1 додавання, 1 квадратний корінь та 2 ділення) для кожної з них. Під час обчислення A_1 в (1.1) кількість множень матриць Ґівенса на сповпчик із двох елементів (4 множення та 2 додавання) дорівнює $(n-1)^2$. Для A_2 кількість таких множень буде $(n-2)^2$, і так далі. Як результат, маємо складність $O(n^3)$:

$$6(n^2-n)/2 + 6\sum_{i=1\ldots n-1}(n-i)^2 = 3n^2 - 3n + 6(n-1)(2n-1)n/6 \approx 2n^3$$

1.2 Рекурсивний QR-розклад

Ідея блочного алгоритму була викладена в [4, 5]. Змінимо порядок дій в алгоритмі 1.1.2 так, що він стане рекурсивним, який у свою чергу містить

інший рекурсивний алгоритм (алгоритм QP). Нехай матриця $M_{2^n \times 2^n}$ є розділеною на чотири рівні блоки: $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$.

1.2.1 Алгоритм рекурсивного QR-розкладу

Алгоритм складається із трьох кроків:

1. Першим кроком є рекурсивне обчислення QR-розкладу блоку C: $Q_1 C = C^U$

$$M_1 = \begin{pmatrix} I & 0 \\ 0 & Q_1 \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} A & B \\ C^U & D_1 \end{pmatrix}, \quad D_1 = Q_1 D \tag{1.3}$$

2. Другим кроком є обнулення “паралелограма”, що складається із двох трикутних блоків: нижньої трикутної частини блоку A та верхньої трикутної частини блоку C^U:

$$M_2 = Q_2 \begin{pmatrix} A & B \\ C^U & D_1 \end{pmatrix} = \begin{pmatrix} A^U & B_1 \\ 0 & D_2 \end{pmatrix} \tag{1.4}$$

3. Третім кроком є рекурсивне обчислення QR-розкладу блоку D_2: $Q_3 D_2 = D^U$

$$R = \begin{pmatrix} I & 0 \\ 0 & Q_3 \end{pmatrix} \begin{pmatrix} A^U & B_1 \\ 0 & D_2 \end{pmatrix} = \begin{pmatrix} A^U & B_1 \\ 0 & D^U \end{pmatrix} \tag{1.5}$$

Як результат маємо:

$$QM = R,\ M = Q^T R,\ \text{де } Q = \mathrm{diag}(I, Q_3) Q_2 \mathrm{diag}(I, Q_1) \tag{1.6}$$

Враховуючи, що перший та третій кроки є рекурсивними, залишилося описати другий крок, а саме алгоритм обнулення “паралелограма”. Цей алгоритм будемо називати QP-розклад.

1.3 Рекурсивний QP-розклад

Шукаємо розклад матриці P : $Q_2 P = P^U$. Матриця $P = \begin{pmatrix} A \\ C^U \end{pmatrix}$ сформована із лівих блоків матриці M_1 (1.3). Кожен із двох блоків, що формують матрицю

$P_{2n\times n}$, поділили на чотири рівні блоки, тобто маємо 8 блоків, включаючи один нульовий блок та два верхні трикутні блоки: e^U та h^U (1.7).

Ми маємо обнулити елементи між верхньою та нижньою діагоналями матриці P, а також саму нижню діагональ. Для цього будемо використовувати матриці Ґівенса, виконуючи обнулення елементів стовпчиків знизу догори, рухаючись послідовно зліва на право.

Але нас цікавить блочний спосіб розкладу. Враховуючи, що матриці ми розглядаємо розміру 2^n, є можливість виконати 4 рекурсивні обчислення QP-розкладу на блоках матриці P у відповідному порядку: $P_{ll} = Q_{ll}\begin{pmatrix} c \\ e^U \end{pmatrix} = \begin{pmatrix} c^U \\ 0 \end{pmatrix}$ (1.8), далі одночасно виконуємо розклад блоків $P_{ul} = Q_{ul}\begin{pmatrix} a \\ c^U \end{pmatrix} = \begin{pmatrix} a^U \\ 0 \end{pmatrix}$ та $P_{lr} = Q_{lr}\begin{pmatrix} f_1 \\ h^U \end{pmatrix} = \begin{pmatrix} f^U \\ 0 \end{pmatrix}$ (1.9),

останніми розкладаємо блоки $P_{ur} = Q_{ur}\begin{pmatrix} d_2 \\ f^U \end{pmatrix} = \begin{pmatrix} d^U \\ 0 \end{pmatrix}$ (1.10). Відповідно ортоганальні матриці Q_{ll}, Q_{ul}, Q_{lr} та Q_{ur} є розміру $n \times n$.

$$P = \begin{pmatrix} A \\ C^U \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \\ e^U & f \\ 0 & h^U \end{pmatrix}, \quad P^U = \begin{pmatrix} A^U \\ 0 \end{pmatrix} = \begin{pmatrix} a^U & b_1 \\ 0 & d^U \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{1.7}$$

$$\bar{P}_{ll} = Q_{ll}\begin{pmatrix} c & d \\ e^U & f \end{pmatrix} = \begin{pmatrix} c^U & d_1 \\ 0 & f_1 \end{pmatrix} \tag{1.8}$$

$$\bar{P}_{ul} = Q_{ul}\begin{pmatrix} a & b \\ c^U & d_1 \end{pmatrix} = \begin{pmatrix} a^U & b_1 \\ 0 & d_2 \end{pmatrix}, \quad \bar{P}_{lr} = Q_{lr}\begin{pmatrix} 0 & f_1 \\ 0 & h^U \end{pmatrix} = \begin{pmatrix} 0 & f^U \\ 0 & 0 \end{pmatrix} \tag{1.9}$$

$$\bar{P}_{ur} = Q_{ur}\begin{pmatrix} 0 & d_2 \\ 0 & f^U \end{pmatrix} = \begin{pmatrix} 0 & d^U \\ 0 & 0 \end{pmatrix} \tag{1.10}$$

Як результат, отримуємо матриці Q_2 (1.4) та P^U (1.7):

$$Q_2 = \bar{Q}_{ur}\bar{Q}_2\bar{Q}_{ll}, \quad Q_2 P = P^U \tag{1.11}$$

де $\bar{Q}_{ur}$, $\bar{Q}_2$, $\bar{Q}_{ll}$ відповідають наступним матрицям:

$$\bar{Q}_{ur} = \mathrm{diag}(I_{n/2}, Q_{ur}, I_{n/2}),\ \bar{Q}_2 = \mathrm{diag}(Q_{ul}, Q_{lr}),\ \bar{Q}_{ll} = \mathrm{diag}(I_{n/2}, Q_{ll}, I_{n/2}), \qquad (1.12)$$

1.3.1 Складність алгоритму QP-розкладу

Загальна кількісь множень матричних блоків розміру $n/2 \times n/2$ дорівнює 28: 8 множень необхідно для матриць $\bar{P}_{ll}$ (1.8) і $\bar{P}_{ul}$ (1.9), та 20 множень необхідно для матриці Q_2 (1.11).

Звідси загальна кількість операці: $C_P(2n) = 4C_P(n) + 28M(n/2)$. Припустимо, що для добутку двох матриць розміру $n \times n$ потрібно γn^β операцій і $n = 2^k$, тоді маємо:

$$C_P(2^{k+1}) = 4C_P(2^k) + 28M(2^{k-1}) = 4^k Cp(2^1) + 28\gamma \sum_{i=0}^{k-1} 4^{k-i-1} 2^{i\beta} =$$

$$28\gamma(n^2/4)\frac{2^{k(\beta-2)} - 1}{2^{(\beta-2)} - 1} + 6n^2 = 7\gamma \frac{n^\beta - n^2}{2^\beta - 4} + 6n^2.$$

$$Cp(n) = \frac{7\gamma n^\beta}{2^\beta(2^\beta - 4)} + \frac{n^2}{2}(3 - \frac{7\gamma}{2^{\beta+1} - 8})$$

1.4 Складність алгоритму QR-розкладу

Давайте оцінимо кількість операцій $C(n)$ в блочно-рекурсивному алгоритмі QR-розкладу. Припустимо, що складність матричного множення $M(n) = \gamma n^\beta$, а складність алгоритму QP-розкладу $C_P(n) = \alpha n^\beta$, де α, β, γ є константами, $\alpha = \frac{7\gamma}{2^\beta(2^\beta - 4)}$ та $n = 2^k$:

$$C(n) = 2C(n/2) + Cp(n) + 6M(n/2) = 2C(2^{k-1}) + Cp(2^k) + 6M(2^{k-1}) =$$

$$C(2^0)2^k + \sum_{i=0}^{k} 2^{k-i} Cp(2^i) + 6\sum_{i=0}^{k} 2^{k-i} M(2^{i-1}) = \alpha \sum_{i=0}^{k} 2^{k-i} 2^{i\beta} + 6\gamma \sum_{i=0}^{k} 2^{k-i} 2^{(i-1)\beta} =$$

$$(\alpha 2^k + 6\gamma 2^{k-\beta}) \sum_{i=0}^{k} 2^{i(\beta-1)} = (\alpha + 6\gamma 2^{-\beta}) \frac{2^\beta n^\beta - 2n}{2^\beta - 2} = \frac{\gamma(2^\beta 6 - 17)(n^\beta - \frac{2n}{2^\beta})}{(2^\beta - 4)(2^\beta - 2)}.$$

Якщо $\beta = 3, \gamma = 1$, маємо складність $(31/24)(n^3 - n/4) < 2n^3$. Якщо $\beta = \log_2 7, \gamma = 1$, то маємо складність $(5/3)(n^\beta - 2n/7) < 1.3n^\beta$.

1.5 Паралельний алгоритм QR-розкладу (DAP)

Алгоритм QR-розкладу було реалізовано в рамках DAP — децентарлізованої паралельної системи управління завданнями програмного пакету MathPartner. Основною вимогою до використання системи є наявність блочно-рекурсивного алгоритму

1.5.1 Програмний пакет Mathpar.DAP

MathPartner — система комп'ютерної алгебри, яка дозволяє виконувати чисельні та символьні обчислення алгебраїчних виразів. Як компонет MathPartner, розвивається система для паралельного виконання завдань із децентралізованим динамічним управлінням — DAP.

Програмний пакет реалізований мовою програмування Java, а система DAP використовує MPI як транспорт для обміну повідомленнями між процесорами. Використовується реалізація MPI OpenMPI із оболонкою для Java.

В цій роботі будуть використовуватися наступні класи із пакету mathpar:

- Element — абстракція для числових елементів (має такі похідні класи, наприклад, NumberZ — клас цілих чисел, NumberR64 — обгортка над типом double, NumberR — клас дійсних чисел будь-якої величини);
- MatrixS — клас розріджених матриць із визначеними операціями, елементами матриці є похідні від класу Element;
- Ring — кільце, в рамках якого виконуються основні операції (наприклад, "Z[]" — операції над цілими числами, "R64[]" — над дійсними із подвійною точністю (double), "R[]" — над числами із заданою точністю: поля Accuracy та MachineEpsilonR).

Основними компонентами системи DAP є:

- DropTask — абстрактний клас завдання. Для реалізації блочно-рекурсивного алгоритму в рамках DAP потрібно створити похідний він нього клас, де описати деталі алгоритму;
- DispThread — диспетчерський потік, відповідає за комунікацію між процесорами, реалізує обмін повідомленнями та розподіляє завдання, від похідних DropTask, між вільними процесорами. Виконується на кожному процесорі;
- CalcThread — рахувальний потік, відповідає за обчислення завдань, похідних від DropTask. Виконується на кожному процесорі.

Як користувачі програмного пакету, маємо страву тільки із описом алгоритму у похідному класі від DropTask. DispThread та CalcThead є ядром системи, що працює з абстрактними завданнями DropTask та розподіляє їх між процесорами, тож користувач позбавлений необхідності реалізовувати такі деталі паралельної програми.

Опишемо більш детально поля та методи DropTask, із якими користувач має справу при реалізації алгоритму. Почнемо із основних:

1) *Element[] inData* — масив вхідних даних;
2) *abstract Element[] inputFunction(Element[] data, int inputKey, Ring ring)* — метод попередньої обробки та підготовки вхідних даних (наприклад, тут можна виконати поділ вхідної матриці на блоки). Отримує на вхід як параметр масив вхідних даних *inData, inputKey* — позначає які дані отримані для вхідної функції для попередньої обробки (1 — повні дані, 2 — основні дані, 3 — додаткові дані), використовується для завдань, що можуть мати додаткові компоненти (наприклад, MultiplyAdd — завдання паралельного множення двох матриць та додавання додаткової матриці до результату $AB + C$);
3) *abstract DropTask[] doAmin( )* — у термінології DAP amin це список підзавдань, похідних від DropTask, що беруть участь у виконанні даного алгоритму (наприклад, Multiply — завдання паралельного множення двох матриць);

4) *int[][] arcs* — двовимірний масив зв’язків між підзавданнями, де кожен одновимірний масив відповідає зв’язкам вихідної функції підзавдання із вхідною функцією іншого завдання. arcs має довжину списку підзавдянь плюс 2: з індексом 0 вихідний масив методу *inputFunction(...)* та на останньому місці порожній масив, що відповідає аргументу *data* методу *outputFunction(...);*
5) *abstract boolean isItLeaf()* — метод, що перевіряє чи є розмір вхідних даних листковим, тобто достатньо малим, щоб виконувати завдання послідовно;
6) *abstract void sequentialCalc(Ring ring)* — метод, для послідовного обчислення завдання;
7) *Element[] outputFunction(Element[] data, Ring ring)* — метод пост-обробки даних, отриманих із підзавдань після завершення виконання алгоритму, зазвичай використовується для збору та формування результату;
8) Element[] outData — масив вихідних даних.

Також є допоміжні поля та методи, що забезпечують коректність виконання завдання:

- *int type* — тип завдання, у кожної реалізації DropTask має бути унікальний тип;
- *int inputDataLength* — кількість вхідних даних (розмір масиву вхідних даних);
- *int numberOfMainComponents* — кількість основних даних, з якими алгоритм може почати виконуватись, із загальної кількості вхідних даних;
- *int numberOfMainComponentsAtOutput* — кількість основних даних після обробки у методі *inputFunction(...)*;
- *int resultForOutFunctionLength* — розмір масиву даних, що повинен надійти як аргумент до методу *outputFunction(...)*;
- *void setLeafSize(int newLeafSize)* — метод для встановлення розміру послідовного обчислення завдання.

1.5.2 Паралельний блочно-рекурсивний алгоритм QR-розкладу

Для реалізації алгоритму QR-розкладу було створено три завдання DropTask:

- QRDecomposition — реалізація в DAP алгоритму QR-розкладу із розділу 1.2, виконується розклад $M = QR$;
- QPDecomposition — реалізація в DAP алгоритму QP-розкладу із розділу 1.3, для зручності виконується розклад $P = Q^T P^U$;
- MultiplyVar — розширення завдання паралельного множення Multiply для більш тонкої підготовки та збирання вхідних даних.

Опишемо кожен блок паралельного QR-розкладу, всі блоки та зв'язки між ними зображені на відповідній схемі (Рис. 1.1):

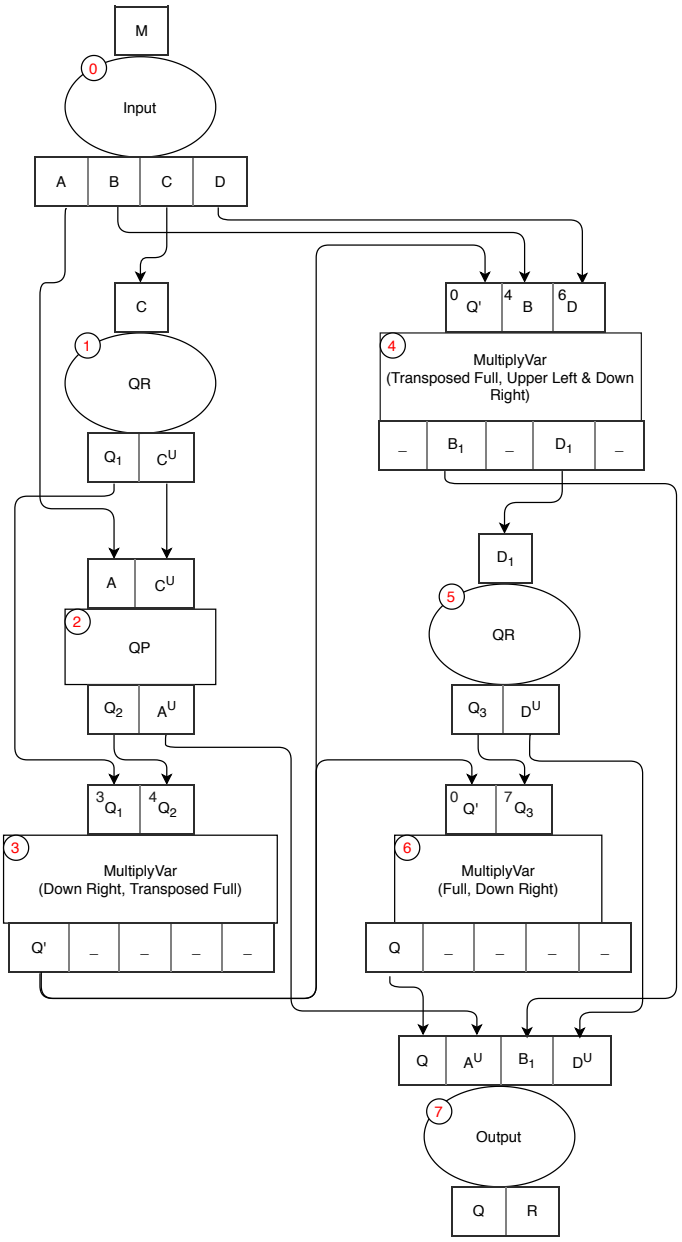

Рисунок 1.1 - Схема QR-розкладу в DAP

(0) вхідна функція — розбиває вхідну матрицю M на чотири блоки;

(1) рекурсивний виклик QR для блоку C (1.3);

(2) обнулення “паралелограма”, виклик QP для блоків A, C^U (1.4);

(3) акумуляція ортоганальних матриць Q_1, Q_2 (1.6), використовується завдання MultiplyVar, що дозволило виконати паралельне блочне множення $\mathrm{diag}(I_{n/2}, Q_1)Q_2^T$;

(4) множення транспонованої акумульованої ортоганальної матиці на праві блоки матриці M. Слід зазначити, що на відміну від алгоритму в (1.3, 1.4) ми не виконували множення на праві блоки, що дозволило скоротити кількість множень, а виконали множення після акумульовання огртоганальних матриць (1.6);

(5) рекурсивний виклик QR для блоку D_1 (1.5);

(6) акумуляція ортоганальної матриці Q_3 (1.6), виконується завдання MultiplyVar, для множення повної матриць на нижню праву чверть;

(7) вихідна функція, виконує збір результатів із блоків.

1.5.3 Паралельний блочно-рекурсивний алгоритм QP-розкладу

Аналогічно до схеми завдання QR-розкладу, опишемо його підзавдання — QP-розклад. Як зазначалося вище, для зручності ми реалізуємо QP-розкладання у наступному вигляді: $P = Q^T P^U$. Це дозволило уникнути транспонувань у самій задачі QP-розкладу, адже там є чотири рекурсивні виклики.

Опишемо кожен блок схеми (Рис. 1.2), де зображено структуру та зв’язки між підзавданнями:

(0) вхідна функція — розбиває кожен із двох вхідних блоків на чотири блоки (1.7);

(1) перший рекурсивний виклик QP для блоків c, e^U (1.8), але на відміну від описаної в алгоритмі без множення Q_{ll} на блоки d, f;

(2) множення Q_{ll} на блоки d, f (1.8) за допомогою MultiplyVar;

(3) другий рекурсивний виклик QP для блоків a, c^U (1.9), без множення Q_{ul} на блоки b, d_1;

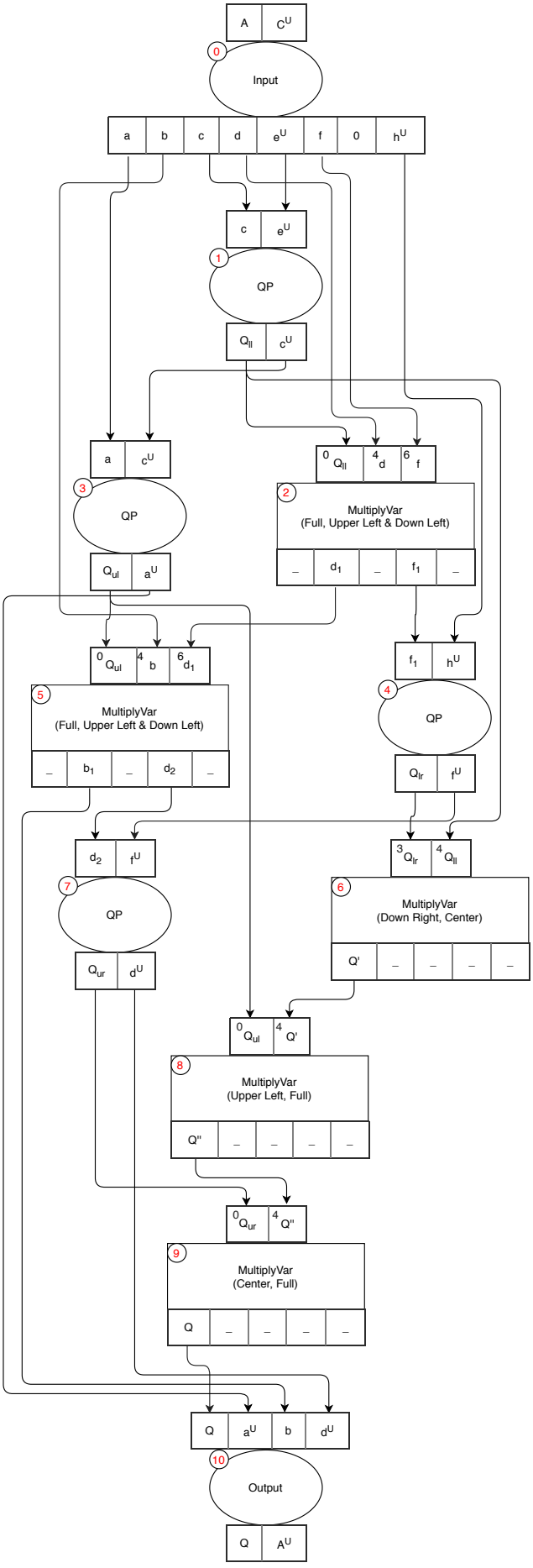

Рисунок 1.2 - Схема QP-розкладу в DAP

(4) третій рекурсивний виклик QP для блоків f_1, h^U (1.9);

(5) множення Q_{ul} на блоки b, d_1 (1.9) за допомогою MultiplyVar;

(6) акумуляція ортоганальних матриць Q_{lr} та Q_{ll} (1.11, 1.12), де Q_{lr} є в правим нижнім блоком, а Q_{ll} — центральним блоком;

(7) четвертий рекурсивний виклик QP для блоків d_2, f^U (1.10);

(8) акумуляція ортоганальної матриці Q_{ul}, що вбудована як правий нижній блок, за допомогою MultiplyVar (1.11, 1.12);

(9) акумуляція ортоганальної матриці Q_{ur}, що вбудована як центральний блок, за допомогою MultiplyVar (1.11, 1.12);

(10) вихідна функція — збирання верхньої трикутної матриці A^U з блоків a^U, b_1 та d^U та формування результату (1.11).

1.5.4 Конфігурація множення в DAP (MultiplyVar)

В DAP вже було реалізовано завдання для паралельного блочного множення Multiply (AB), MultiplyAdd ($AB + C$) та інші. Але не було задань, що могли б виконувати множення матриць сформованих із окремих блоків, які можуть бути отримані як результат виконання інших завдань. Якщо реалізовувати перевизначення функції підготовки вхідних даних (*DropTask.inputFunction*) для кожної окремої ситуації (1.5.2 пункти (3), (4), (6) та 1.5.3 пункти (2), (5), (6), (8), (9)), то було б багато похідних класів від завдання Multiply, які можуть бути некорисні у повторному використанні для реалізації інших завдань, крім QR-розкладу. Із метою задовольнити потреби у множенні такого виду матриць, було реалізовано розширення Multiply — завдання MultiplyVar, що має більші можливості налаштування для формування та обробки вхідних блоків матриць.

Завдання MultiplyVar розширює клас MultiplyAdd (Рис. 1.3) та перевизначає методи *inputFunction* та *outputFunction* для попередньої обробки вхідних даних та пост-обробки результатів відповідно до конфігурації завдання. Конфігурація (MultiplyVarConfig) реалізована у форматі шаблону будівельник (builder pattern). Далі наведено всі можливості для конфігурації завдання:

1. конфігурація складових компонентів. Умовно є два блоки, кожен із блоків можна зробити від'ємним:
   a. обов'язковий блок множення AB — складається із двох матриць;
   b. опційний блок додавання C — відповідно складається із однієї матирці.
2. конфігурація кожної окремої матриці $A, B, (C)$:
   a. встановити діагональний елемент матриці (нуль або одиниця) у порожніх (невстановлених) блоках;
   b. встановити блок як цілу матрицю;
   c. встановити центральний блок: $\mathrm{diag}(El_{n/2}, M, El_{n/2})$, де El діагональна матриця із певним діагональним елементом;
   d. встановити блоки як певні чверті матриці (1.13);
   e. транспонувати будь-який матричний блок чи саму матрицю;
   f. встановити копію певного блоку на будь-яку позицію із попередніх пунктів (щодо позицій див. опис нижче).

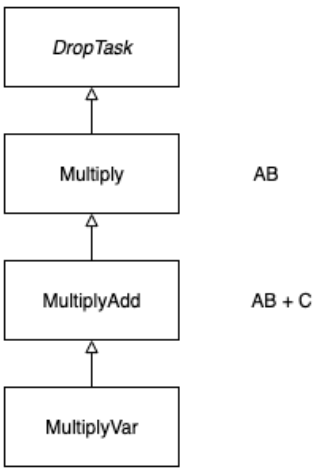

Рисунок 1.3 - Діаграма класу MultiplyVar в DAP

Масив вхідних даних зазнав змін: замість 2 матриць як в Multiply, або 3 матриць як в MultiplyAdd, завдання MultiplyVar має 8 або 12 вхідних матриць в залежності чи є опційне додавання. Відповідно для кожної матриці відводиться по 4 місця. Якщо матриця повна чи є центральним блоком, то на вхід подається позицію із індексом 0 для матриці A, матриці B, C розміщуються із зміщенням 4 та 8 відповідно. Блоки в чвертях розміщуються за формулою

$$a)\ \bar{M}_1 = \begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} \quad b)\ \bar{M}_2 = \begin{pmatrix} M_1 & 0 \\ 0 & M_4 \end{pmatrix} \quad c)\ \bar{M}_3 = \begin{pmatrix} M_1 & 0 \\ M_3 & 0 \end{pmatrix} \tag{1.13}$$

$index = block\_index + shift - 1$, де $block\_index$ це індекс блоку (1.13 а), $shift$ це зміщення матриці ($shift(A) = 0$, $shift(B) = 4$, $shift(C) = 8$).

Масив вихідних даних має 5 матриць, на 0 позиції результат, далі чотири блоки матриці результату відповідно до індексів (1.13 а).

Із кодом можна ознайомитись за посиланням.

2 Прямий алгоритм обчислення SVD

Розглянемо факторизацію матриці A на добуток трьох матриць: ортоганальної матриці U^T, тридіагональної матриці T та ортоганальної матриці V^T, тобто UTV-розклад: $UAV = T$, еквівалентно $A = U^T T V^T$, адже $U^{-1} = U^T$, $V^{-1} = V^T$. Розглядаємо розклад квадратної матриці над полем дійсних чисел.

2.1 Матриці Ґівенса для обнулення справа

Визначимо матрицю Ґівенса для обнулення елементів справа. Розглянемо частковий випадок для матриці 2×2. Тоді щоб обнулити елемент β рівняння буде мати такий вигляд:

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} a & 0 \\ c & d \end{pmatrix}, \text{ або відповідно } \quad \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a & 0 \\ c & d \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

де числа s та c задовольняють нерівність $s^2 + c^2 = 1$. Якщо $\beta = 0$, тоді $c = 1$, $s = 0$. Якщо $\beta \neq 0$, тоді $c = \frac{\beta}{\Delta}$, $s = \frac{\alpha}{\Delta}$, де $\Delta = \sqrt{\alpha^2 + \beta^2}$. Таку матрицю надалі будемо позначати як $g_{\alpha,\beta}$.

Аналогічно до 1.1.1 визначимо матрицю Ґівенса для обнулення елементів справа для довільного розміру. Нехай маємо матрицю B, (i, j) та $(i, j+1)$ елементами якої є α та β, і всі елементи над ними є нульовими: $\forall(s < i) : (b_{s,j} = 0)\ \&\ (b_{s,j+1} = 0)$. Тоді матриця буде мати такий вигляд: $\bar{G}_{i,j} = \text{diag}(I_{j-1}, g_{\alpha,\beta}, I_{n-j-1})$. Результат добутку $B\bar{G}_{i,j}$ відрізняється від A двома колонками j та $j+1$, всі елементи над рядком i залишаються нульовими, а елемент $b_{i,j+1}$ стає нульовим. Поєднання властивостей матриць $G_{i,j}$ та $\bar{G}_{i,j}$ дає можливість сформувати наступний алгоритм.

2.2 Матриці Ґівенса для алгоритму UTV-розкладу

Для обнулення елементів будемо використовувати матриці Ґіванса $G_{i,j}$ та $\bar{G}_{i,j}$. Аналогічно до алгоритму QR-розкладу будемо послідовно обнулювати стовпчики, але на відміну від QR-розкладу до піддіагоналі. Подібно до

обнулення стовпчиків будемо виконувати обнулення рядків матрицями $\bar{G}_{i,j}$, виконуючи множення справа.

(1) Почнемо із лівого сповпця і обнулимо в ньому всі елементи, що розташовані під другим рядком, а у верхньому рядку обнулимо всі елементи, що розташовані справа від другого стовпчика:

$$A_1 = G_{2,1}G_{3,1}\ldots G_{n-2,1}G_{n-1,1}A\bar{G}_{1,n-1}\bar{G}_{1,n-2}\ldots\bar{G}_{1,3}\bar{G}_{1,2} \tag{2.1}$$

(2) Далі обнулимо елементи у другому стовпчику, що розташовані під третім рядком, та елементи у другому рядку, що розташовані справа від третього стовпчика:

$$A_2 = G_{3,2}G_{4,2}\ldots G_{n-2,2}G_{n-1,2}A_1\bar{G}_{2,n-1}\bar{G}_{2,n-2}\ldots\bar{G}_{2,4}\bar{G}_{2,3} \tag{2.2}$$

(k) Визначимо $G_{[k]} = G_{k+1,k}, G_{k+2,k}\ldots G_{n-2,k}G_{n-1,k}$ та $\bar{G}_{[k]} = \bar{G}_{k,n-1}\bar{G}_{k,n-2}\ldots\bar{G}_{k,k+2}\bar{G}_{k,k+1}$, де $k = 1,2,\ldots,n-1$. Тоді для обнулення k-го стовпчика та рядка потрібно обчислити наступний добуток:

$$A_k = G_{[k]}A_{k-1}\bar{G}_{[k]}$$

(n-2) Вкінці обчислення обнуляємо елементи $a_{n-1,n-2}$ та $a_{n-2,n-1}$:

$$A_{n-2} = G_{[n-2]}A_{n-3}\bar{G}_{[n-2]} = G_{n-1,n-2}A_{n-3}\bar{G}_{n-2,n-1}$$

Оцінимо кількість операцій в алгоритмі. Необхідно обчислити матриць Ґівенса $2\times 2n^2 - n - 2(n-1) = n^2 - 3n + 1$, а також виконати 6 операцій на кожну (2 піднесення до квадрату, 1 додавання, 1 корінь квадратний, 2 ділення). Під час обчислення A_1 в (2.1) кількість множень матриць Ґівенса 2×2 на стовпчик (зліва) або рядок (справа) із двох елементів (4 множення та 2 додавання) $2(n-1)(n-2)$. Для A_2 (2.2) кількість таких множень буде $2(n-2)(n-3)$, і так далі. Як результат маємо складність $O(n^3)$:

$$6(n^2 - 3n + 1) + 6\sum_{i=1\ldots n-2} 2(n-i)(n-i-1) = 6n^2 - 18n + 6 + 6(\frac{1}{3}n^3 - n^2 + \frac{2}{3}n) \approx 2n^3$$

2.3 Обчислення UTV-розкладу на n^2 процесорах

Була спроба розробити блочно-рекурсивний алгоритм для UTV-розкладу для подальшої паралелізації цього алгоритму за допомогою DAP. Але виявилося, що реалізація блочно-рекурсивного алгоритму UTV-розкладу не є ефективною і потребує чималої кількості повторних обнулень елементів, що “затираються”. Проте ідея прямого паралельного алгоритму UTV-розкладу є актуальною, адже це суттєво наближає нас до обчислення SVD. Було розроблено паралельний алгоритм на n^2 процесорах. Основна ідея полягає у розбитті вхідної матриці A на n^2 кадратних матричних блоків та обробки кожного блоку на окремому процесорі.

Розглянемо приклад розкладу матриці $A_{m\times m}$ n^2 процесорах, при $n = 2$ (2.3). Тобто розбиваємо матрицю A на чотири блоки. Обнулення починається із 1 сповпчика та рядка (2.1), але як видно на (2.3), є складнощі із обнуленням елементів $a_{k+1,1}$ та $a_{1,k+1}$, адже елементи задіяні в їх обнуленні ($a_{k,1}$ та $a_{1,k}$, відповідно) перебувають в інших блоках. Для цього після обнулення $k-1$ елемента у сповпчику будемо виконувати передачу першого рядка у блоці до верхнього блоку, аналогічно, після обнулення $k-1$ елемента у рядку будемо передавати перший стовпчик до лівого блоку.

$$A_{m\times m} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m-1} & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m-1} & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,m-1} & a_{m-1,m} \\ a_{m,1} & a_{m,2} & \cdots & a_{m,m-1} & a_{m,m} \end{pmatrix} => \begin{matrix} A_1 = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{pmatrix} & A_2 = \begin{pmatrix} a_{1,k+1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{k,k+1} & \cdots & a_{k,m} \end{pmatrix} \\ A_3 = \begin{pmatrix} a_{k+1,1} & \cdots & a_{k+1,k} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,k} \end{pmatrix} & A_4 = \begin{pmatrix} a_{k+1,k+1} & \cdots & a_{k+1,m} \\ \vdots & \ddots & \vdots \\ a_{m,k+1} & \cdots & a_{m,m} \end{pmatrix} \end{matrix} \quad (2.3)$$

де $k = m/n$.

2.3.1 Схема алгоритму

Щоб краще пояснити алгоритм перейдемо від матричного до схематичного представлення. Розглянемо розклад довільної квадратної матриці $A_{m\times m}$ (2.3), але розбитої на 16 блоків. На схемах (Рис. 2.1 та Рис. 2.2) зображено

процес обнулення сповпчика та рядка, що належать лівому блочному стовпчику (блоки 0, 4, 8, 12) та верхньому блочному рядку (блоки 0, 1, 2, 3).

Означимо умовні позначення в схемі:

1. пронумеровані квадрати позначають одночасно матричні блоки $k \times k$ (2.3) та процесори, що їх оброблюють;
2. червоним кольором позначені активні блоки, де відбувається обнулення (Рис. 2.1 a — блоки 3 та 12);
3. заповнений вертикальний прямокутник (стовпчик) позначає матрицю Ґівенса $G_{ik,j}G_{ik+1,j}\ldots G_{ik+k,j}$ розміру $k \times k$ (Рис. 4a — зліва від блоків 12, 13, 14, 15) або матрицю $g_{\alpha,\gamma}$ розміру 2×2 (Рис. 4b — зліва від блоків 8, 9, 10, 11). Червоний колір позначає блок, де матриця згенерована; Стрілка від цього прямокутника до блоку або блоку з додатковим рядком позначає множення на блок або на матрицю $2 \times k$, утворену останнім рядком блоку та додатковим рядком. Далі будемо називати такі прямокутники матрицями удару справа;
4. заповнений горизонтальний прямокутник (рядок) позначає матриці Ґівенса $\bar{G}_{i,jk}\bar{G}_{i,jk+1}\ldots\bar{G}_{i,jk+k}$ розміру $k \times k$ (Рис. 2.1 a — зверху від блоків 3, 7, 11, 15), або матрицю $g_{\alpha,\beta}$ розміру 2×2 (Рис. 2.1 b — зверху від блоків 2, 6, 10, 14). Червоний колір позначає, де матриця знегерована. Стрілка від цього прямокутника до блоку або блоку з додатковим сповпчиком позначає множення на блок або на матрицю $k \times 2$, утворену з останнього сповпчика блоку та додаткового сповпчика. Далі будемо називати такі прямокутники матрицями удару зліва;
5. порожній вертикальний прямокутник позначає додатковий стовпчик $k \times 1$, що є лівим стовпчиком правого блоку (Рис. 2.1 a — справа від блоків 2, 6, 10, 14);
6. порожній горизонтальний прямокутник позначає додатковий рядок $1 \times k$, що є верхнім рядком нижнього блоку (Рис. 2.1 a — знизу від блоків 8, 9, 10, 11);

7. порожній малий квадрат позначає один додатковий елемент, що є нульовим елементом (1,1) наступного діагонального матричного блоку;
8. стрілка сірого кольору позначає передачу рядка/сповпчика/елемента до іншого блоку;
9. стрілка жовтого кольору позначає поверняння рядка/сповпчика/елемента до свого блоку;
10. стрілка рожевого кольору позначає повернення рядка/сповпчика із додатковим елементом до свого блоку;
11. пунктирна стрілка від матриць удару зліва/справа позначає розповсюдження згенерованої матирці удару (червоного кольору) до інших блоків, для подальшого множення;
12. стрілка рожевого кольору від матриць удару 2×2 позначає множення із виключенням 0 рядка при множенні справа або 0 сповпчика при множенні зліва.

2.3.2 Алгоритм UTV-розкладу на n^2 процесорах

Відповідно до (2.1) починаємо обнулювати стовпчики знизу зліва та рядки зверху справа, тобто спочатку обнулення відбувається на крайніх активних блоках — в цьому прикладі 12 та 3 блоки. Нехай будемо обнулювати i-й сповпчик та рядок матриці $A_{m\times m}$ (2.3), при $1 \le i \le k$, де $k = m/n, n = 4$.

Перший етап. Паралельно виконуємо обнулення сповпчика в 12 блоці та рядка в 3 блоці (Рис. 2.1 а).:

1. Після обнулення сповпчика (в 12 блоці) маємо матрицю $G_{m-k+1,i}\dots G_{m,i}$ яку множимо на блок та передаємо всім блокам, що розташовані справа. Потім передаємо перший рядок блоку як додатковий до верхнього блоку процесору зверху (процесор 8);
2. Після обнулення рядка (в 3 блоці) маємо матрицю $\bar{G}_{i,m}\dots\bar{G}_{i,m-k+1}$, яку множимо на блок та передаємо всім блокам, що розташовані нижче. Потім передаємо перший сповпчик блоку як додатковий до лівого блоку (процесор 2);

3. Коли процесори справа (блоки 13, 14) отримують матриці удару зліва, то виконують множення на блоки, та передають перший рядок блоку до процесора зверху;
4. Коли процесори знизу (блоки 7, 11) отримують матриці удару справа, то виконують множення на блоки, та передають перший стовпчик блоку до процесора зліва;
5. Для діагональних блоків (блок 15) порядок передачі додаткових сповпчиків/рядків/елементів відрізняється від описного у двох попередніх пунктах: передача сповпчика, рядка та елемента відбувається вже після отримання матриць удару зліва та справа, та їх відповідного множення на блок.

Другий етап. Після виконання обнулення на 3 та 12 процесорах (Рис. 2.1 a), активними стають наступний процесорний сповпчик та рядок (Рис. 2.1 b). Виконуємо обнулення елемента в додатковому рядку (8 процесор) та додатковому сповпчику (2 процесор):

1. Після обнулення елемента у додатковому рядку (8 процесор), множимо отриману матрицю удару зліва розміру 2×2 на матрицю розміру $k \times 2$, утворену із останнього рядка блоку та додаткового рядка. Далі передаємо матрицю удару всім процесорам, розташованим справа (процесори 9, 10, 11), та повертаємо додатковий рядок до нижнього блоку (12 процесор);

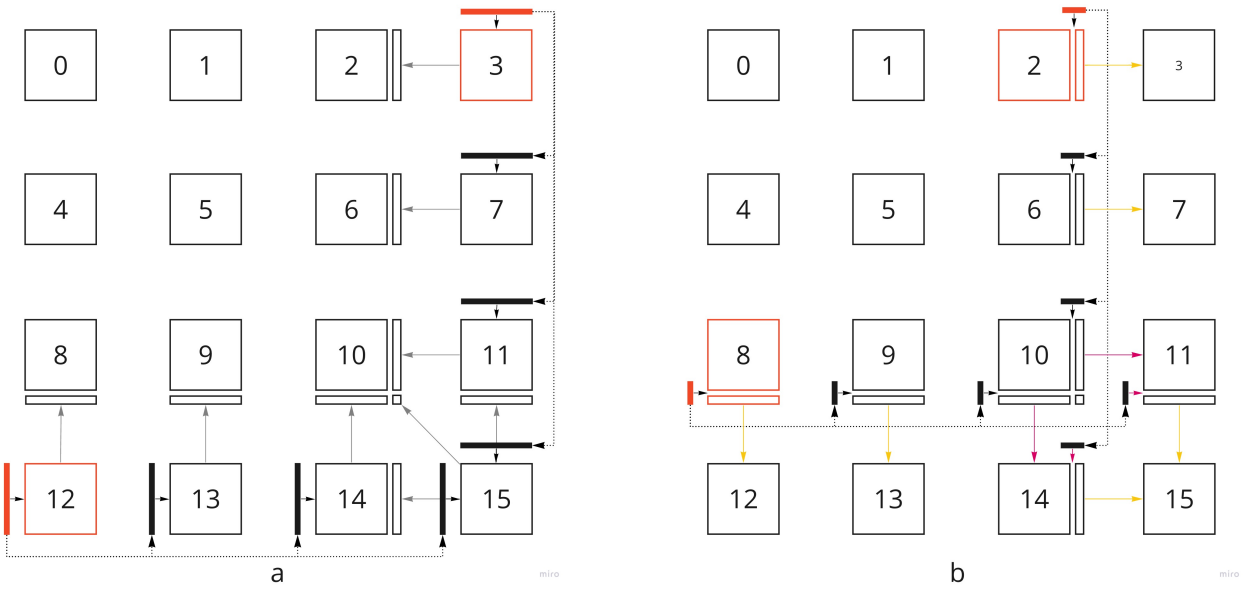

Рисунок 2.1 - Схема UTV-розкладу на n^2 процесорах

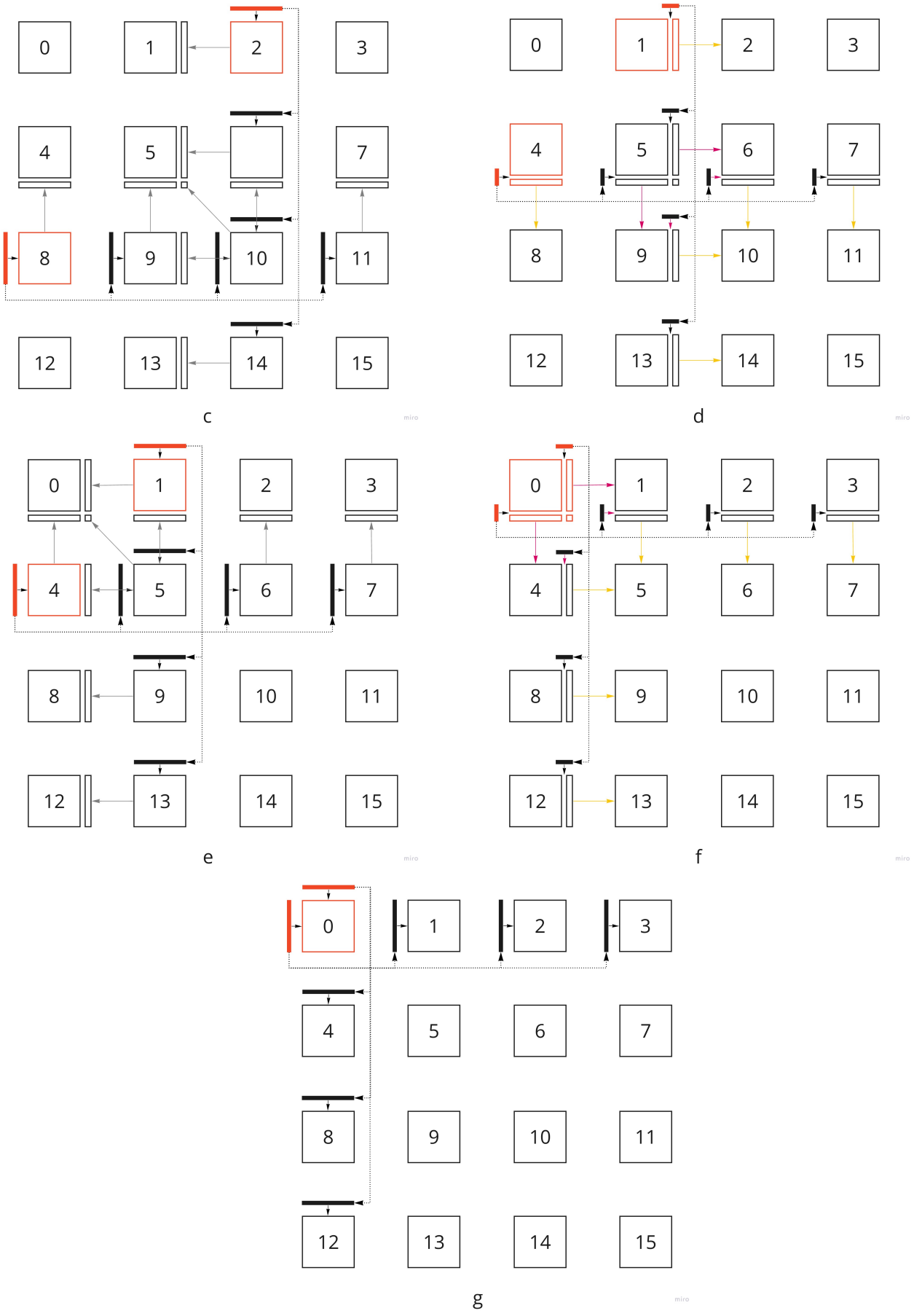

Рисунок 2.2 - Схема UTV-розкладу на n^2 процесорах (продовження)

2. Після обнулення елемента у додатковому стовпчику (2 процесор), множимо отриману матрицю удару справа розміру 2×2 на матрицю розміру $2 \times k$, утворену із останнього сповпчика блоку та додаткового сповпчика. Далі передаємо матрицю удару всім процесорам, розташованим знизу (процесори 6, 10, 14), та повертаємо додатковий стовпчик до правого блоку (3 процесор);
3. Після отримання матриці удару справа 2×2 (процесори 6), виконуємо множення на матирцю $2 \times k$, утворену із останнього стовпчика блоку та додаткового стовпчика, та повертаємо додатковий стовпчик до правого блоку;
4. Для діагонального блоку (процесор 10), аналогіно до попереднього етапу, множення та порядок повернення додаткових стовпчика, рядка та елемент відрізняється від попередніх пунктів: після того, як отримали матрицю удару зліва 2×2, виконуємо множення на матрицю $2 \times (k+1)$, утворену із доповненням справа до матирці $2 \times k$ вектор-сповпця 2×1 із останнього елемента додаткового сповпчика та додаткового елемента. Після того, як отримали матрицю удару справа 2×2, виконуємо множення на матрицю $(k+1) \times 2$, утворену із доповненням останнього рядка до матирці $k \times 2$ вектора 1×2 із останнього елемента додаткового рядка та додаткового елемента. Після виконання обох множень, повертаємо до нижнього блоку (процесор 14) додатковий рядок з додатковим елементом, а до блоку справа (процесор 11) додатковий стовпчик із додатковим елементом;
5. Множення на матрицю удару зліва/справа 2×2 для блоків, що розташовані під діагональним блоком та справа від діагонального блоку відрізняється від множення в попарадніх пунктах. Матрицю удару зліва 2×2 множимо на матрицю $2 \times (k-1)$, утворену із нижнього рядка блоку та додаткового рядка, починаючи із другої колонки. Матрицю удару справа 2×2 множимо на матрицю $(k-1) \times 2$, утворену із останнього стовпчика блоку та додаткового стовпчика, починаючи із другого рядка. Далі аналогічно до попередніх пунктів повертаємо рядок до нижнього блоку або сповпчик до

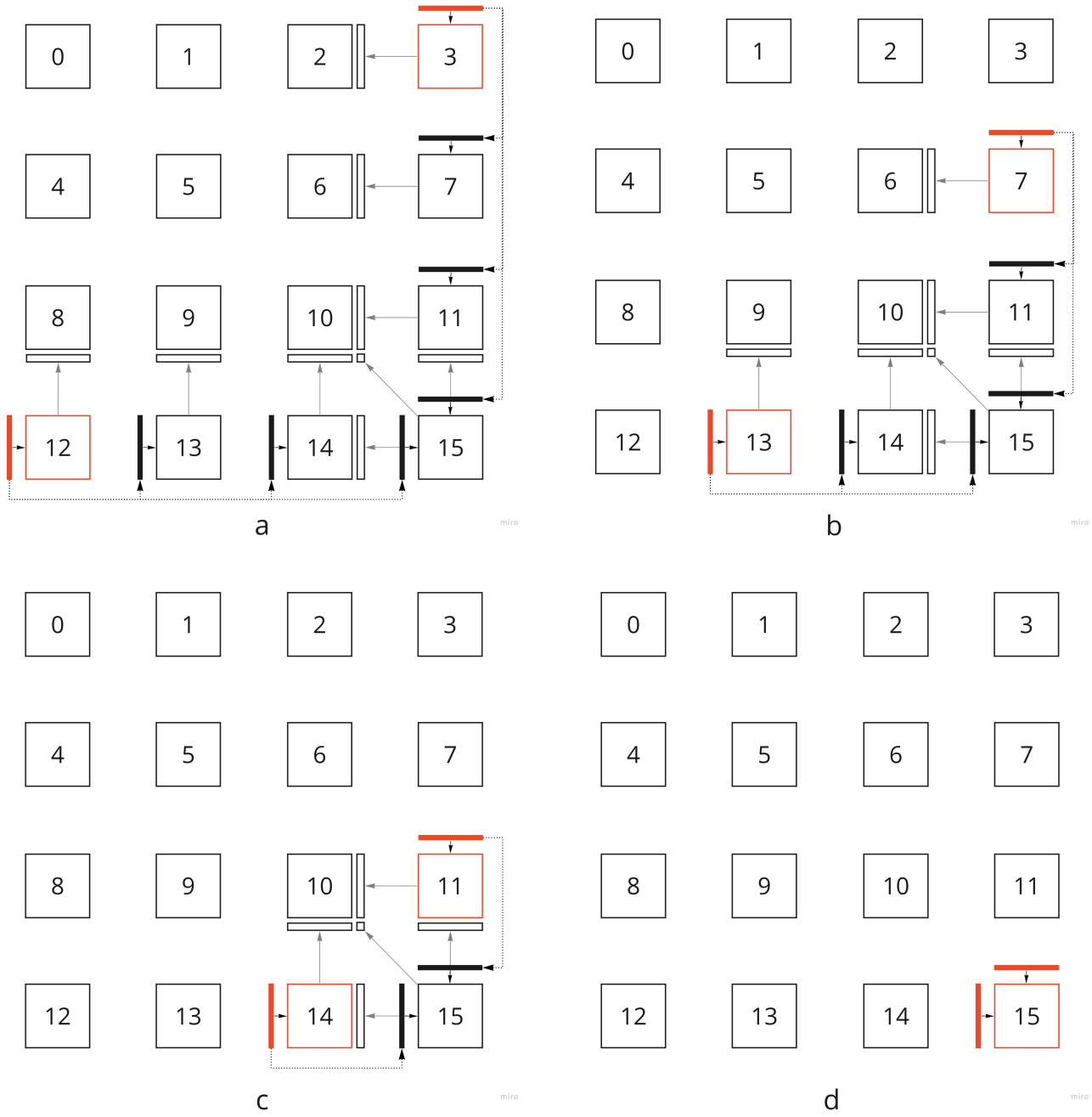

Рисунок 2.3 - Схема UTV-розкладу на n^2 процесорах. Ітераційне зміщення

правого блоку. Тут варто зауважити, що це реалізовано, щоб кутовий елемент діагонального блоку повернувся назад від попереднього діагонального блоку;

Далі виконуємо знову етап 1, для $n-1$ процесорного стовпчика (блоки 2, 6, 10, 14) та рядка (блоки 8, 9, 10, 11).

Другий та перший етапи послідовно виконуються над кожним j-тим процесорним рядком та сповпчиком (Рис. 2.1 b; Рис. 2.2 c, d, e), $j = n-1, n-2 \ldots 3,2$. Для n-го процесорного рядка та сповпчика виконується

тільки перший етап (Рис. 4a) , а для першого процесорного рядка та стовпчика — тільки другий етап (Рис. 2.2 f) та видозміна першого етапу, яка полягає у обнуленні елементів рядка та сповпчика діагонального блоку до тридіагонального вигляду (Рис. 2.2 g). Так відбувається обнулення одного стовпчика та рядка матриці квадратної A, розділеної на n^2 блоків між n^2 процесорами.

Таким чином виконуємо обнулення кожного i-го рядка та стовпця матриці $A_{m\times m}$, де$1 \le i \le m-2$. Кожні k таких обнулень, що еквівалентно приведенню діагонального блоку (Рис. 4 — блоки 0, 5, 10, 15) активного процесорного рядка та стовпчика до тридіагонального вигляду, процес обнулення зміщується на наступний процесорний рядок та сповпчик (Рис. 2.3). В результаті маємо тридіагональну матрицю T, розташовану на діагональних блоках: блоки 0, 5, 10, 15 (елементи із сусідніх блоків, що утворюють піддіагональ та наддіагональ перадаються при останній ітерації на блоці).

2.3.3 Обчислення ортогональних матриць UTV-розкладу

В розділі 2.3.3 описано розклад матриці $A_{m\times m}$ до тридіагональної матриці $T_{m\times m}$, тож залишилося обчислити та зібрати ортогональні матиці $U_{m\times m}$ та $V_{m\times m}$.

Після кожного обнулення блоку (2.3.3 перший етап — пункти 1, 2; другий етап — пункти 1, 2), матриці удару зберігалися на процесорах, де відбувалося обнулення. Обчислення кожної із ортогональних матриць можна виконати паралельно на n процесорах, тож збираємо всі накопичені матриці удару зліва та справа на першому процесорному сповпчику та рядку відповідно (Рис. 2.4 a). Але враховуючи особливість, що на 0 процесорі буде всього $k-1$ множення матриць удару відповідної матриці, то можемо використовувати спільно його як для обчислення U так і V (Рис. 2.5). Варто зауважити, що обчислення можна починати вже після обнулення перших k сповпчиків та рядків матриці A, адже далі $2n-1$ у першому процесорному рядку та стовпчику будуть вільні (Рис. 2.3 b).

Для обчислення матриці U на процесорному стовпчику генеруємо акумуляційні блоки — матричні блоки розміру $k \times m$, що відповідають одиничній матриці $I_{m \times m}$, поділеної на n горизонтальних блоків розміру $k \times m$. Опишемо послідовность кроків обчислення:

1. Отримуємо додатковий перший рядок акумуляційного блоку нижче, виконуємо множення матриці удару 2×2 на матрицю $2 \times m$, утворену цим рядком та останнім рядком поточного акумуляційного блоку, повертаємо додатковий рядок до акумуляційного блоку нижче;
2. Виконуємо множення матриці удару, що відповідає за обнулення відповідного сповпчика блоку, на акумуляційний блок;
3. Відсилаємо перший рядок акумуляційного блоку до процесора вище.

Для останнього блоку не виконуємо пункт 1 (Рис. 2.5 — процесор 12), а для першого блоку не виконуємо пункт 1 (Рис. 2.5 — процесор 0). Після обчислення всіх акумуляційних блоків, передаємо їх на 0 процесор, де вони будуть зібрані та буде сформований результат виконання алгоритму UTV-розкладу (Рис. 2.4 b).

Аналогічно відбувається обчислення ортогональної матриці V. На процесорному рядку (Рис. 2.5 — блоки 0, 1, 2, 3) генеруємо акумуляційні блоки

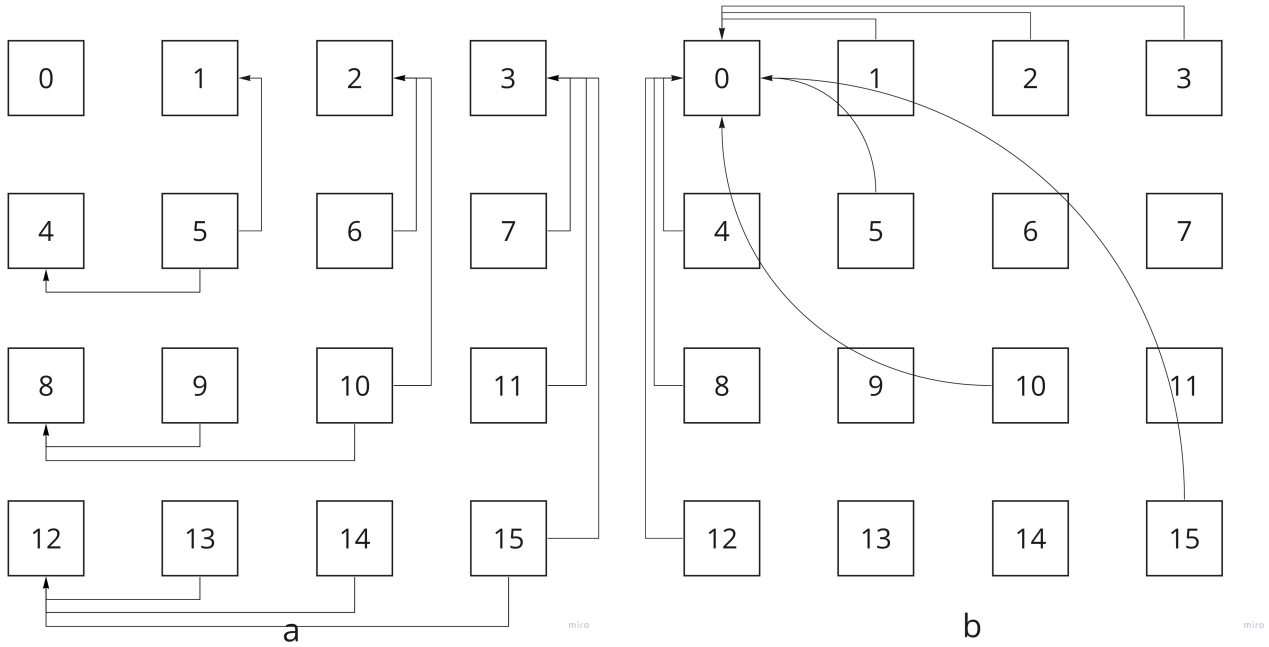

Рисунок 2.4 - Схема збору матриць удару для обчислення (a) та збору результатів (b)

розміру $m \times k$, що відповідають одиничній матирці $I_{m \times m}$, поділеної на n вертикальних блоків розміру $m \times k$. Виконуємо наступну послідовність кроків:

1. Отримуємо додатковий перший стовпчик акумуляційного блоку права, виконуємо множення матриці удару 2×2 на матрицю $m \times 2$, утворену цим стовпчиком та останнім стовпчиком поточного акумуляційного блоку, повертаємо додатковий стовпчик до акумуляційного блоку справа;
2. Виконуємо множення справа матриці удару, що відповідає за обнулення відповідного рядка блоку, на акумуляційний блок;
3. Відсилаємо перший стовпчик акумуляційного блоку до процесора зліва.

Для останнього блоку не виконуємо пункт 1 (Рис. 2.5 — процесор 3), а для першого блоку не виконуємо пункт 1 (Рис. 2.5 — процесор 0). Після обчислення всіх акумуляційних блоків, передаємо їх на 0 процесор, де вони будуть зібрані та буде сформований результат виконання алгоритму UTV-розкладу (Рис. 2.4 b).

Після обнулення кожного діагонального блоку, результати також передаються до 0 процесора (Рис. 2.4 b).

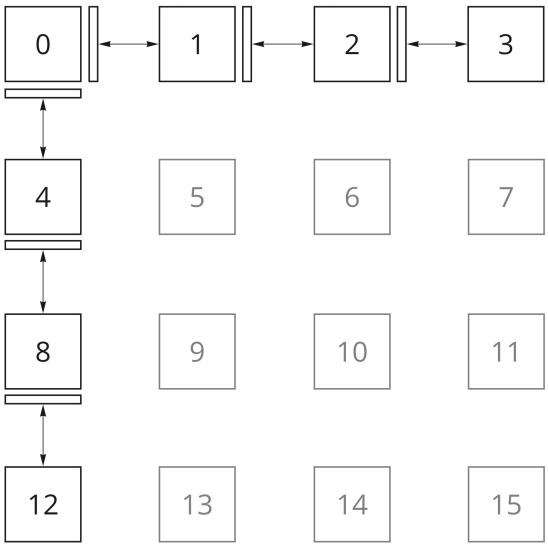

Рисунок 2.5 Схема обчислення матриць U та V

2.3.4 Програмна реалізація UTV-розкладу на n^2 процесорах

Алгоритм UTV-розкладу було реалізовано в рамках програмного пакету MathPartner, мовою програмування Java. Було використано наступні класи із програмного пакету:

- *Ring* — кільце, в рамках якого виконуються операції;
- *MatrixS* — клас розріджених матриць, що також реалізує матричні операції.

Із кодом можна ознайомитись за посиланням.

2.3.4.1 Абстракція процесора

Із алгоритму, описаного в 2.3.2, природньо виділяється абстракція (клас) Processor, що відповідає за обробку певного блоку вхідної матриці. За операціями, що мають виконувати кожен процесор над блоком, можна виділити три класи таких процесорів:

- *LeftProcessor* — відповідає за обробку певного блоку, що розташований зліва від діагонального блоку;
- *DiagonalProcessor* — відповідає за обробку певного діагонального блоку;
- *RightProcessor* — відповідає за обробку певного блоку, що розташований зліва від діагонального блоку.

Перелічені класи є розширенням абстрактного класу Processor, де реалізований спільний функціонал.

Відповідно до алгоритму обчислення ортогональний матриць описаного в 2.3.3, реалізовано розширення класів, перелічених вище:

LeftAccumulationProcessor, DiagonalAccumulationProcessor, RightAccumulationProcessor.

Ідея використання абстрактного процесора виглядає привабливою, адже дозволяє за рангом реального процесора створити абстрактний відповідно до схеми, зображеної на рисунку 2.6, а потім запустити його на виконання. Діаграму класів наслідування класу *Processor* зображено на рисунку А.1.

Також із класу *Processor* виділяються або є компонентами наступні класи (див. рисунок А.2):

Рисунок 2.6 - Класи процесорів відповідно до рангу

- *Coordinator* — відповідає за навігацію даного процесора серед інших процесорів;
- *Communicator* — відповідає за передачу повідомлень іншим процесорам. Використовує абстракцію *Transport* для відправки та отримання повідомлень. Реалізовано різні транспорти: *LocalTransport* використовується для відладки, емулюючи передачу повідомлень на одному процесорі послідовно, *MPITransport* використовує MPI для передачі повідомлень. Ми використовуємо OpenMPI;
- *MatrixStorage* — накопичує матриці удару після обнулення на кожному процесорі, які потім передаються на відповідний процесор для обчислення.

Процесори-акумулятори (наступні класи: *LeftAccumulationProcessor, DiagonalAccumulationProcessor, RightAccumulationProcessor)* містять відповідний клас, що відповідає саме за обчислення ортогональних матриць U, V — *MatrixAccumulator*. А клас *DiagonalAccumulationProcessor* ще відповідає за збір всіх матриць із акумульованих блоків та формування

результату (Рис. 2.4 b), для цього виділено клас *MatrixCollector* із відповідними реалізаціями для кожної із матриць U, T, V.

2.3.4.2 Стани процесорів

Кожен процесор у будь-який момент часу перебуває у певному стані зображеному на рисунку 2.7. При цьому залежно від стану, в якому він перебуває, та від типу процесора, він виконує певні дії відповідно до алгоритму описаного в 2.3.2 та 2.3.3. Детальніше про стани, зображені на рисунку 2.7:

1. Початковий — стан, в якому відбувається розповсюдження вхідних даних: кожен процесор отримує блок вхідної матриці. Цей процес зображено на рисунку 2.4 a;
2. Пасивний — процесор отримує матриці удару, виконує відповідні множення на матричний блок, але не виконує обнулення (рисунок 2.1 a — блоки 13, 14, 15, 7, 11 та інші не активні);
3. Активний — стан, в якому виконується обнулення стовпчика/рядка, та пов’язані з цим дії (рисунок 2.1 a — блоки 0, 1, 2, 3, 4, 8 та 12);

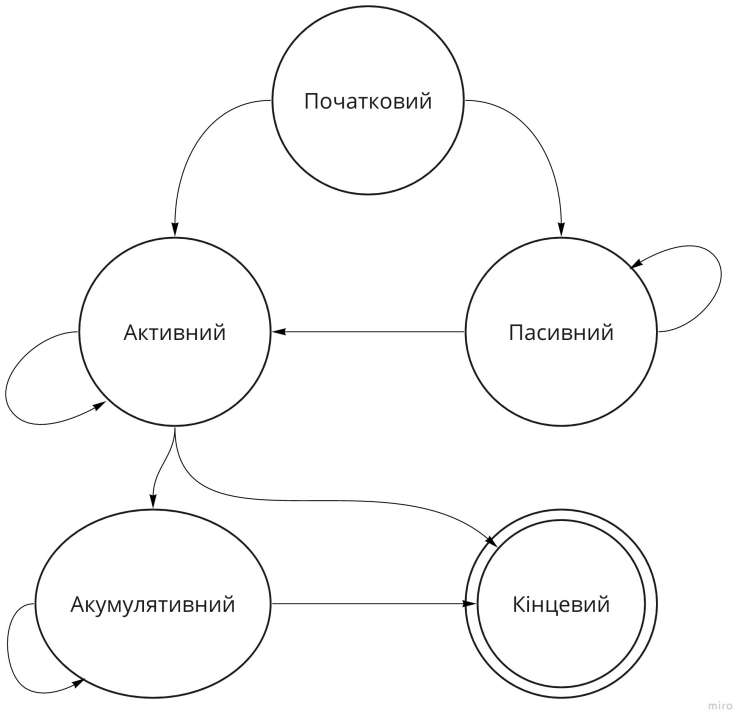

Рисунок 2.7 - Стани процесорів

4. Акумулятивний — стан, в якому відбувається обчислення ортогональних матриць U, V зображене на рисунку 2.5, відповідно перебувати в цьому стані можуть тільки процесори-акумулятори;
5. Кінцевий — термінальний стан процесора.

2.4 Обчислення UTV-розкладу на n процесорах

Описана в розділі 2.3 схема обчислення UTV-розкладу на n^2 процесорах має певні недоліки. Зі схеми, зображеної на рисунках 2.1-2.3, зрозуміло, що одночасно в обчисленні тридіагональної матриці T задіяні щонайбільше $2n-1$ процесори, а з рисунку 2.5, при обчисленні ортогональних матриць U та V задіяні також $2n-1$ процесор. Тобто схема обчислення UTV-розкладу на n^2 процесорах не є ефективною через погану завантаженість процесорів.

Розглядалися варіанти реалізації UTV-розкладу на $2n-1$ або $2n$ процесорах, де n процесорів відповідають за блочні рядки вхідної матриці, а інші n процесорів відповідають за блочні стовпчики вхідної матриці, але це вимагає дублювання обчислень. Тому ми прийшли до схеми UTV-розкладу на n процесорах, де кожен процесор відповідає за n блоків вхідної матриці.

2.4.1 Схема алгоритму UTV-розкладу на n процесорах

Схему виконання UTV-розкладу зображено на рисунку 2.8. Одним кольором позначено віртуальні процесори, що будуть запущені на виконання на одному реальному процесорі.

Схема має ще одну відмінність він схеми із розділу 2.3 — об'єднання передачі матриць удару розміру 2×2 та $k \times k$ (фактично реалізовано як $k-1$ матриця розміру 2×2). Тобто дії зображені на рисунку 2.2 c, d еквівалентні діям, зображеним на рисунку 2.8 c.

2.4.2 Програмна реалізація

Щоб реалізувати цей варіант алгоритму на n процесорах, було доповнено програмну реалізацію схеми для n^2 процесорів (див. рисунок Б.1):

- *CompositeProcessor* — абстракція процесора, яка запускає на виконання віртуальні процесори *Processor*. Враховуючи, що віртуальні процесори мають виконуватися по черзі, дії в кожному стані процесора розділені на такти. Такти розмежовуються отриманнями повідомлень через розширений комунікатор *CompositeCommunicator;*
- *CompositeCommunicator* — розширення комунікатора, що задовольняє обмін повідомленнями між локальними (розташованими на тому ж реальному процесорі) віртуальними процесорами — використовується *LocalTransport*, та

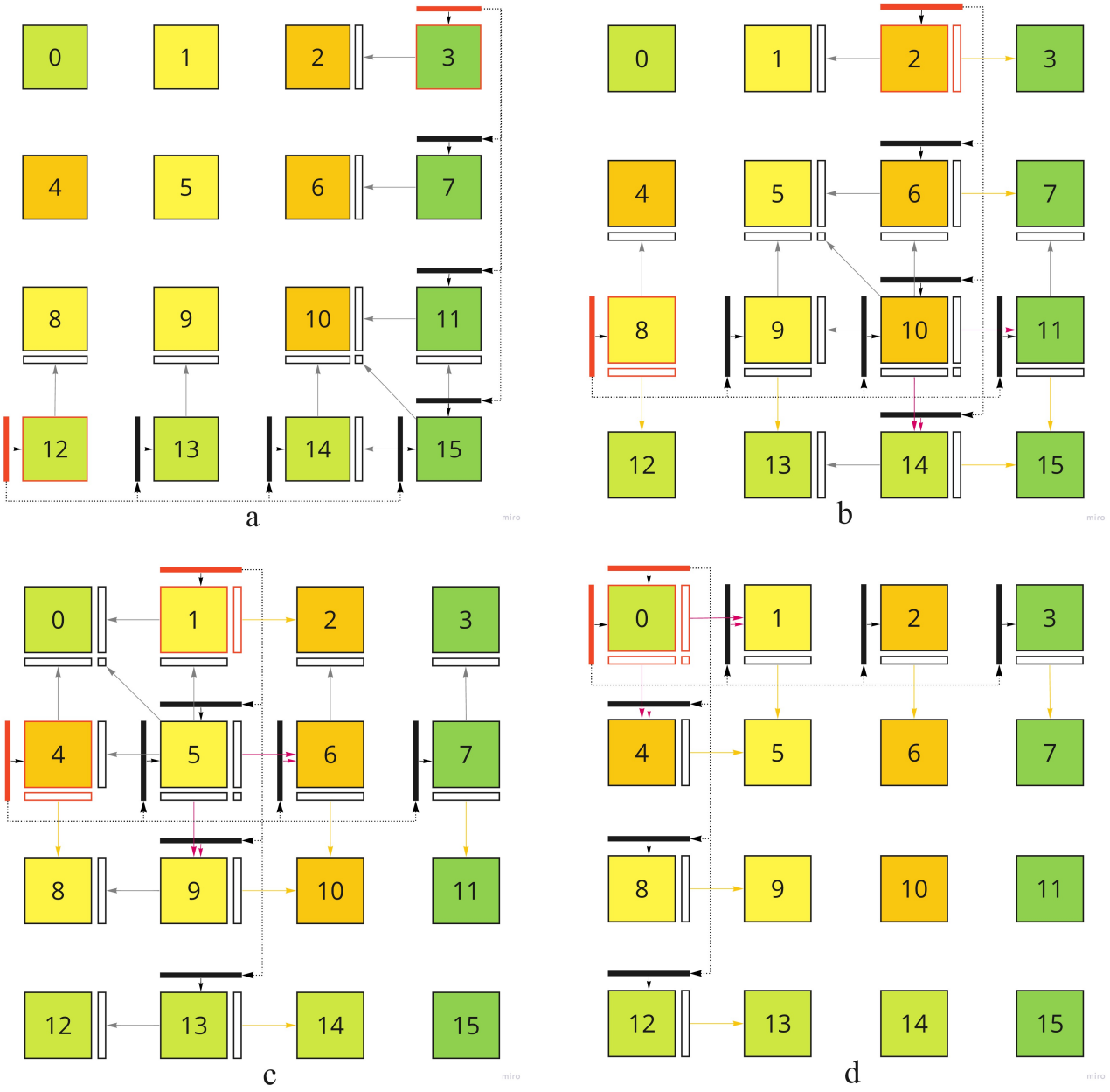

Рисунок 2.8 - Схема UTV-розкладу на *n* процесорах

між віртуальними процесорами, розташованими на різних реальних процесорах — *MPITransport*.

Порядок виклику віртуальних процесорів також є важливим.

2.4.3 Проблеми в реалізації

В MPITransport використовуються відправлення повідоблення з блокуванням та отримання повідомлення із блокування. Що є джерелом дедлоків при передачі великих повідомлень в рамках саме схеми для n процесорів. Як варіант вирішення цієї проблеми може бути реалізований механізм рукостискань — запит на підтвердження готовності отримати повідомлення.

Реалізація колективних операцій (наприклад, scatter для початкової розсилки вхідних даних) для передачі великих повідомлень, тобто великих матриць, має свої особливості. Адже може відбуватися переповнення буферів обміну при відправленні повідомлення. Рішенням є використання інтра-комунікаторів, для відправки повідомлень, що не перевищують об’єм буфера, підгрупам процесорів.

3 Результати тестування на кластері

В цьому розділі наведено результати тестувань на кластері реалізованих алгоритмів із розділів 1 та 2.

Всі тести були виконані на кластері Joint Supercomputer Center of the Russian Academy of Sciences. Дякуємо за допомогу в організації можливості проведення експериментів співробітників Лабораторії алгебраїчних обчислень Тамбовського Державного Університету.

3.1 Результати тестування QR-розкладу

Перед початком тестування паралельного блочно-рекурсивного алгоритму QR-розкладу, реалізованого в рамках DAP, треба визначити листковий розмір — розмір даних, з яким завдання перестають ділитися на менші. У системі DAP завдання з листковим розміром може бути передано на інший процесор, для обчислення. Таке тестування варто проводити перед початком основного тестування, адже для різного апаратного забезпечення (процесор, мережа) можуть бути різні листкові розміри.

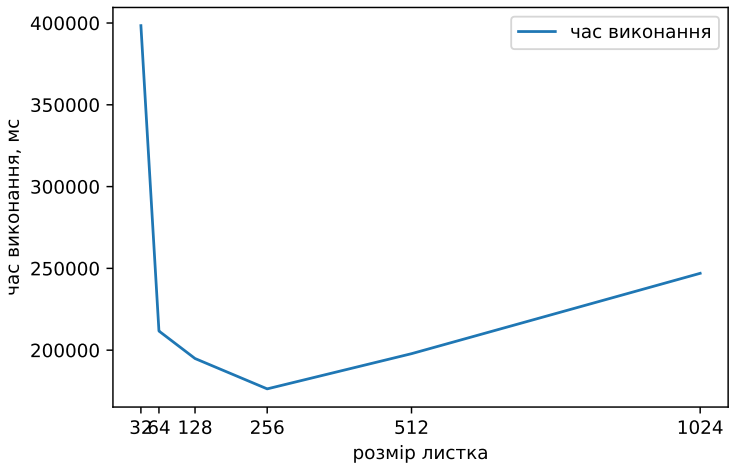

Рисунок 3.1 - Графік залежності часу виконання від листкового розміру

З метою визначення найкращого листкового розміру було здійснено серію тестів QR-розкладу на 16 процесорах для матриц розміру 2048×2048 із різним листковим розміром. Результати наведено в таблиці 3.1. Виявилося, що найкращий час виконання при листковому розмірі 256. Як видно на рисунку 3.1,

Процесори	Розмір даних	Листковий розмір	Час виконання, мс
16	2048	32	398398
16	2048	64	211698
16	2048	128	194894
16	2048	256	176331
16	2048	512	197852
16	2048	1024	246984

Таблиця 3.1 - Результати тестів для пошуку найкращого листкового розміру

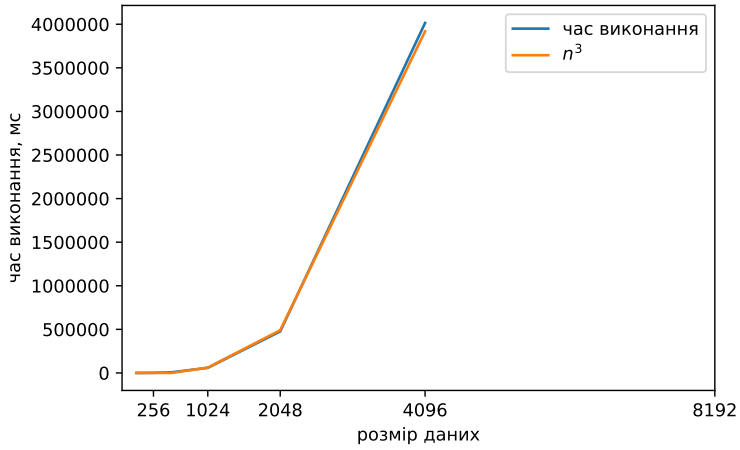

Рисунок 3.2 - Графік для послідовного обчислення QR-розкладу

на розмір 256 припадає мінімум, тож всі наступні тести паралельного алгоритму QR-розкладу будуть наведені із відповідним листковим розміром задачі.

Також виконаємо тестування послідовного блочно-рекурсивного алгоритму QR-розкладу, щоб мати змогу обчислити прискорення при паралельному виконанні. Результати тестування наведено в таблиці 3.2. Графік залежності часу виконання від розміру вхідних даних зображено на рисунку 3.2, як видно із рисунку складність послідовного алгоритму відповідає $O(n^3)$.

Паралельні тести виконували на вхідних даних розміру $m \times m$, де $m = 512, 1024, 2048, 4096$, на 4, 8, 16 та 32 процесорах. Результати представлено в таблиці 3.3. Графік залежності часу виконання від розміру

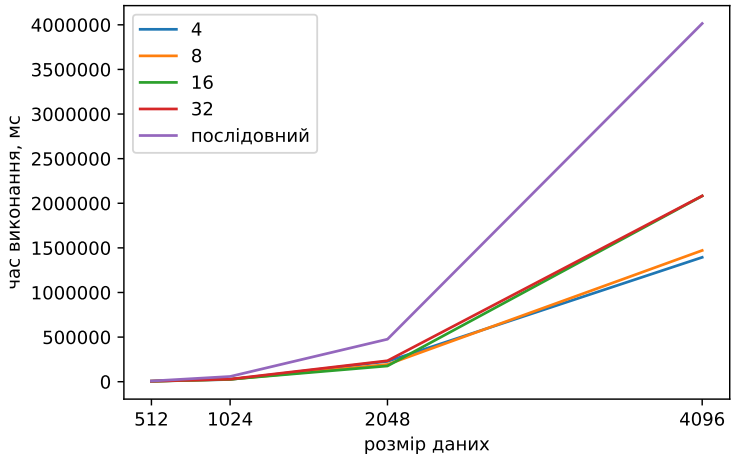

Рисунок 3.3 - Графік для паралельного обчислення QR-розкладу

вхідних даних зображено на рисунку 3.3, також наведено порівняння із послідовним виконанням. Максимальне прискорення було 2.88 при 4 процесорах та розмірі даних 4096×4096.

Відсутність збільшення прискорення при збільшенні кількості процесорів можна пояснити специфікою децентралізованого управління в системі DAP, а саме використання листкового розміру як обмеження поділу задачі, що в свою чергу обмежує кількість задіяних процесорів залежно від співвідношення розміру даних до листкового розміру. На великих розмірах даних спостерігається більше прискорення, що потребує подальшого тестування.

Розмір даних	Час виконання, мс
16	20
32	22
64	106
128	537
256	1627
512	7654
1024	58482
2048	476317
4096	4014704

Таблиця 3.2 - Результати послідовного обчислення QR-розкладу

Процесори	Розмір даних	Час виконання	Прискорення
4	512	5451	1.40
4	1024	24836	2.35
4	2048	227797	2.09
4	4096	1394054	2.88
8	512	5475	1.40
8	1024	26432	2.21
8	2048	193220	2.47
8	4096	1471189	2.73
16	512	5761	1.33
16	1024	27339	2.14
16	2048	176331	2.70
16	4096	2081584	1.93
32	512	5948	1.29
32	1024	28781	2.03
32	2048	235026	2.03
32	4096	2082762	1.93

Таблиця 3.3 - Результати паралельного обчислення QR-розкладу

3.2 Результати тестуваня UTV-розкладу

Перед початком тестування паралельної реалізації алгоритму UTV-розкладу, проведемо тестування послідовної реалізації, щоб мати змогу обчислити прискорення при паралельному виконанні.

Результати наведено таблиці 3.4 зображено на графіку рисунку 3.4. Як видно із результатів послідовного обчислення, реалізація потребує покращення, адже складність алгоритму UTV-розкладу із використанням матриць Ґівенса є $O(n^3)$, а на рисунку 3.4 видно, що реальний час виконання зростає значно швидше.

Тестування паралельної реалізації виконувалося на матрицях розміру $m \times m$, де $m = 32, 64, 128, 256, 512, 1024, 2048$, та на n^2 процесорах, де $n = 2, 4, 8$. Результати представлені в таблиці 3.5. Можна зробити висновок, що

Розмір даних	Час виконання, мс
16	18
32	42
64	270
128	2066
256	27822
512	364558
1024	5218852
2048	74777851

Таблиця 3.4 - Результати послідовного обчислення UTV-розкладу

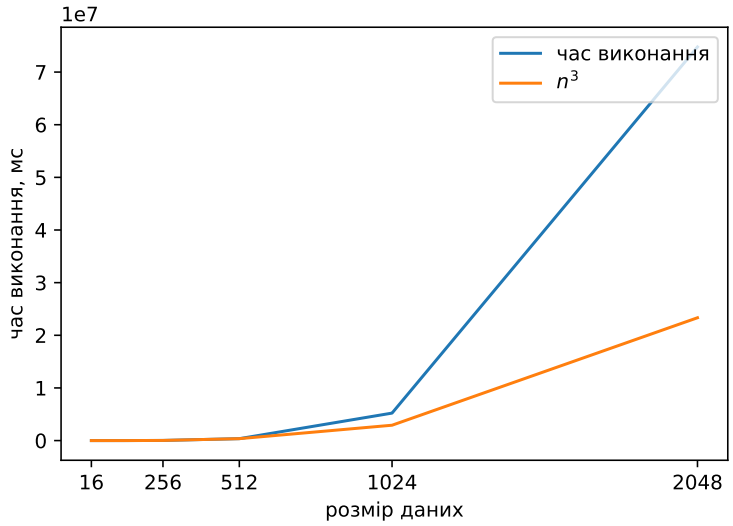

Рисунок 3.4 - Графік послідовного обчислення UTV-розкладу

матриці менші за 256×256 краще розкладати послідовно. Відповідні результати тестів зображено на графіку на рисунку 3.5.

Ми спостерігаємо прискорення при збільшенні кількості процесорів. Із результатів видно, що найбільше прискорення в 15.32 разів при розмірі даних 2048×2048 на 64 процесорах. Враховуючи можливість розвитку моделі алгоритму UTV-розкладу із n^2 до n процесорів, це потребує подальшого дослідження, адже має потенціал до розвитку.

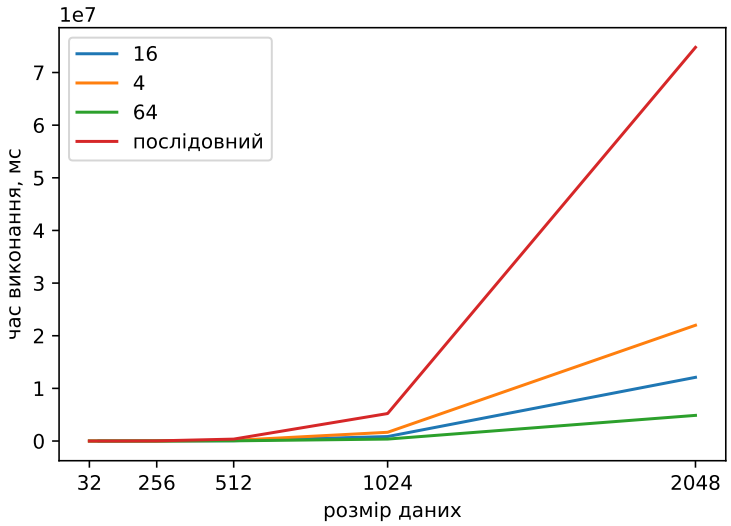

Рисунок 3.5 - Графік паралельного обчислення UTV-розкладу на n^2 процесорах

Процесори	Розмір даних	Час виконання, мс	Прискорення
4	32	299	0.14
4	64	414	0.65
4	128	1109	1.86
4	256	9443	2.95
4	512	123386	2.95
4	1024	1659317	3.15
4	2048	21989949	3.40
16	64	3224	0.08
16	128	3209	0.64
16	256	9448	2.94
16	512	65536	5.56
16	1024	850458	6.14
16	2048	12098699	6.18
64	128	4135	0.50
64	256	7991	3.48
64	512	34118	10.69
64	1024	368643	14.16
64	2048	4871681	15.35

Таблиця 3.5- Результати паралельного обчислення UTV-розкладу

Висновки

Під час ввикоання роботи було дослідження матричні алгоритми факторизації матриць на основі матриць Ґівенса: QR-розклад та UTV-розклад.

Для QR-розкладу показано, що існує блочно-рекурсиний алгоритм, що дозволило реалізувати його паралельний варіант в рамках програмного пакету MathPartner.DAP. Проведено дослідження прискорення рекурсивного алгоритму QR-розкладу, що показало максимальне прискорення в 2.88 рази на 4 процесорах порівняно із послідовним виконанням. Хоча дана реалізація алгоритму має невелике прискорення, це компенсується малим накопичення похибки: при тестуванні максимального розміру матриці 4096×4096 похибка не перевищює точність double. Тестування показує високу маштабованість даної реалізації алгоритму.

Для UTV-розкладу було реалізовано паралельний алгоритм на основі матриць Ґівенса. Спочатку було розроблено варіант для n^2 процесорів, та досліджено прискорення: максимальне прискорення становило в 15.32 разів на 64 процесорах порівняно із послідовним виконанням. Тестуавння показувало гарне прискорення та маштабованість. Подальший аналіз цього алгоритму виявив, що одночасно були навантажені максимум $2n - 1$ процесори із n^2 процесорів. В результаті було запропоновано приведення схеми для n^2 процесорів на схему n процесорів, із розширенням попередньої програмної реалізації, що потребує подальшого дослідження.

Спикок використаних джерел

1. G.I. Malaschonok. MathPartner computer algebra // Programming and Computer Software. 2017. vol. 43:2. pp. 112–118.

2. Малашонок Г. І., Сідько А. А. Розподілені обчислення: ДАП-технологія розпаралелювання рекурсивних алгоритмів. Наукові записки НаУКМА. Комп’ютерні науки. 2018. Том 1. с. 25 – 32.

3. Малашонок Г. І. Хмарна математика MathPartner у Києво-Могилянській академії // Наукові записки НаУКМА. 2017. Том 198. с. 27 – 35.

4. Schönhage A.: Unitre Transformationen groer Matrizen, Numerische Mathematik. 20, pp. 409-417 (1973)

5. G. Malaschonok, "Recursive Matrix Algorithms, Distributed Dynamic Control, Scaling, Stability," 2019 Computer Science and Information Technologies (CSIT), Yerevan, Armenia, 2019, pp. 112-115, doi: 10.1109/CSITechnol. 2019.8895255.

6. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner // НаУКМА. 2020. с. 53-58.

7. Gene H. Golub, Charles F. Van Loan. Matrix Computations – fourth edition // Johns Hopkins University Press. 2013. 784 p.

Додаток А

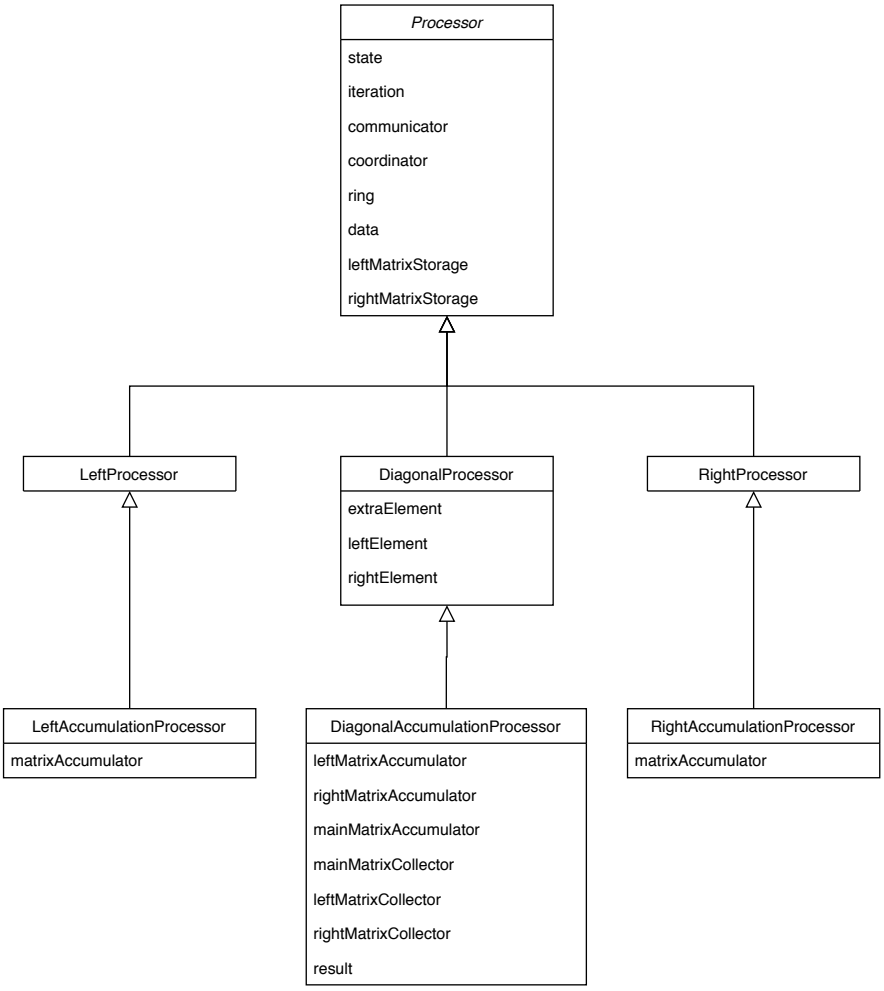

Рисунок 1 - Наслідування класу *Processor*

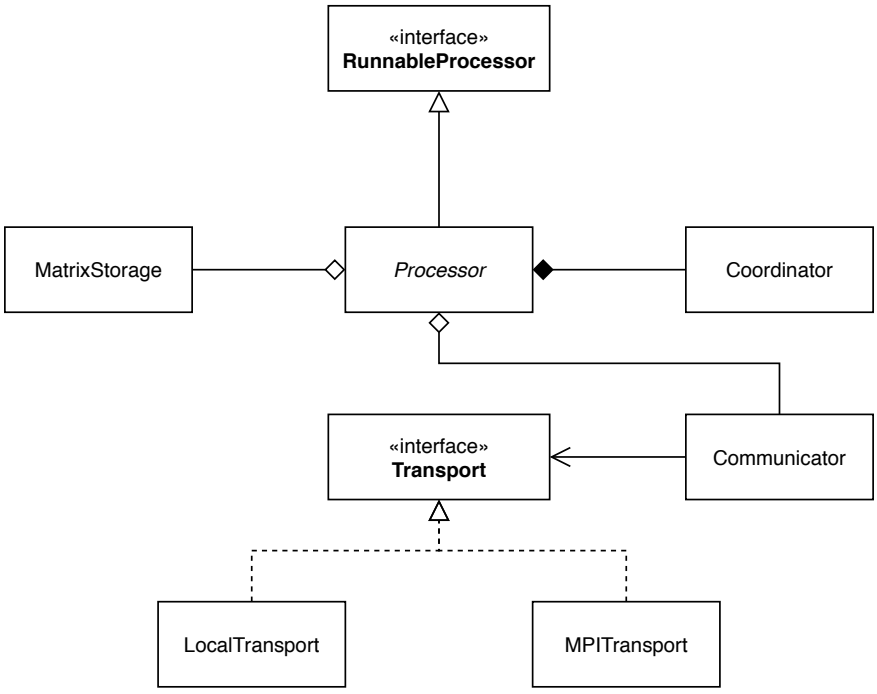

Рисунок 2 - Залежності класу *Processor*

Додаток Б

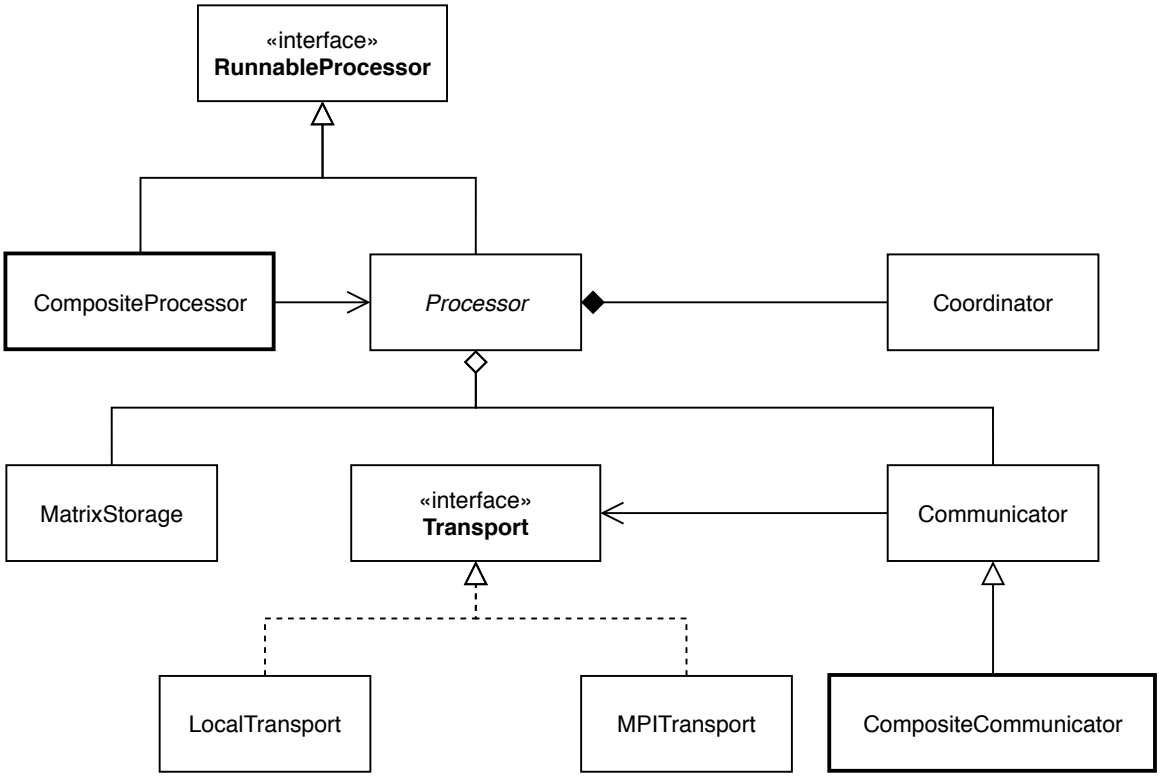

Рисунок 1 - Залежності класу *Processor*. Розширення