Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

Курсова робота

освітній ступінь – бакалавр

на тему: «**Обчислення оберненої матриці на суперкомп'ютері**»

Виконав: студент 4-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Студента Молодцова Філліпа

Керівник Сідько А.А.

магістр прикладної математики, асистент

«16» травня 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри мережних технологій

Малашонок Г. І.

“10” жовтня 2021 року

ЗАВДАННЯ
ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Молодцову Філліпу

1. Тема роботи **«Обчислення оберненої матриці на суперкомп’ютері»**, керівник роботи Сідько Алла Анатоліївна, магістр прикладної математики, асистент

2. Строк подання студентом роботи 20 травня 2022

3. План роботи

Анотація

Вступ

Розділ 1. Опис алгоритму обчислення оберненої матриці

1.1 Означення
1.2 Знаходження оберненої матриці через приєднану
1.3 Обчислення приєднаної матриці
1.4 Блочний алгоритм обчислення приєднаної матриці
1.5 Граф блочного алгоритму обчислення приєднаної матриці
1.6 Блочний алгоритм множення матриць на 4 процесорах

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2021			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтня 2021 – 2 листопада 2021			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2021			
4.	Написання розділів роботи	2 листопада 2021 – 01 березня 2022			
5.	Проміжний контроль виконання роботи	01 лютого 2022			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2022 – 29 березня 2022			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2022			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2022			
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2022			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2022 – 06 травня 2022			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	20 травня 2022			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

Графік узгоджено 10 жовтня 2021 р.

Науковий керівник Сідько Алла Анатоліївна

Виконавець курсової роботи Молодцов Філліп

ЗМІСТ

Анотація

У данній роботі розглядається алгоритм обчислення приєднаної матриці для знаходження оберненої матриці. Найбільше уваги приділено розробці багатопроцесорної програми з метою вирішення даної задачі, на основі блочного алгоритму. Результатом роботи є імплементація вищезгаданого алгоритму за допомогою засобів стандартної бібліотеки Java та в системі динамічного децентралізованого управління завданнями DAP. Були проведені експерименти та зроблено висновки щодо роботи програми та майбутніх вдосконалень.

Ключові слова: алгоритм, вхідні, вихідні данні, результат, список, процеси, багатопроцесорне програмування, паралельне програмування, обчислення приєднаної матриці, алгебраїчні доповнення, обернення, множення, рекурсія, блочний алгоритм, дроп, амін, пайн, повідомлення, граф, арки, розсилка, повідомлення, Java.

Вступ

За більше ніж 70 років існування, галузь використання запрограмованих машин сильно розвинулася та стала широко застосованою. Хоча в попередньому столітті переважало існування програм, що використовували лише один потік для виконання обчислень, уже тоді виникла необхідність у обробці великих масивів даних у найшвидший час. Це можна досягти роздрібленням даних на менші частини та обчисленні за допомогою складної багатопроцесорної системи, яким є суперкомп'ютер. Ще у дев'яностих роках просліджувалася тенденція розповсюдження більшої кількості суперкомп'ютерів, що відрізнялися обчислювальними здібностями, архітектурою та ціною, із появою більшої кількості задач та їхнім ускладненням[1]. Зараз спостерігається продовження цієї тенденції та її загострення, оскільки виник великий попит на обробку колосальних об'ємів даних та на вирішення складних задач з областей машинного навчання та обробки великих даних.

Одним з найефективніших методів універсалізації обчислень великих об'ємів даних – представлення даних у вигляді масивів або матриць та застосування до них обчислень лінійної алгебри. Користь від такого методу обчислення можуть отримати представники галузей метеорології, біології, фармацевтики, епідемології, космології, розділи фізики. Найбільшим завданням є розподілити масиви даних таким чином по процесорам суперкомп'ютеру, щоб обчислення змогли виконуватися якнайближче до центрального процесора, у кеші. Це можна зробити якщо роздрібнити вхідні дані таким чином, щоб вони вмістились у кеші[2].

Досліджуючи тему курсової роботи, було взято до уваги блочні алгоритми. Їхня суть полягає в тому, що застосовується рекурсія для зменшення блоків матриць до потрібного розміру. При тому під час обчислення розгортається

дерево суб-обчислень, яке зберігає операції над матричними блоками у вигляді вузлів. Вхідні дані передаються від кореневого вузла до тих, що виконують однопоточне обчислення невеликих матричних блоків (так званих листків). Коли операції над матричними блоками закінчуються, тоді результати передаються з листків до кореневого вузла, з'єднуючи частини вхідної матриці й формуючи кінцевий розв'язок.

Однією з найбільших задач у цій області – оптимізувати імплементацію блочних алгоритмів для неоднорідних та розріджених даних, які найчастіше трапляються у реальних випадках. Наприклад, Відомий DDP (Differential Data Processing) метод розроблений Е. А. Ільченко [3,4] найефективніше впорається лише з однорідними матрицями, бо метод не передбачає перерозподілу підзадач в залежності від того наскільки швидко один процесор звільниться від даного йому обчислення. Метод, розроблений Г. І. Малашонком та А. А. Сідько, спирається на опрацюваннях Е. А. Ільченка, але при цьому в методі концентрується увага на ефективному обчисленні неоднорідних та розріджених матричних блоків [5,6,7]. На основі цього методу була створена програма децентралізованого управління розподіленими обчисленнями DAP, що вирішує завдання, які спираються на блочних алгоритмах.

Метою роботи є дослідження та аналіз блочного алгоритму обчислення приєднаної матриці та його реалізація за допомогою структури даних системи DAP, а також проведення експериментів над цією реалізацією.

Було виокремлено наступні завдання дослідження:

- Ознайомитися з теорією лінійної алгебри, пов'язаної з обчисленням оберненої матриці

- Проаналізувати алгоритм обчислення оберненої матриці включно з блочним різновидом
- Дослідити структуру даних системи DAP та проаналізувати роль складових у вирішенні задач, що спираються на блочні алгоритми
- Дослідити та описати функціонал різновидів потоків, що використовуються для роботи системи
- Розширення системи для імплементації блочного алгоритму обчислення приєднаної матриці
- Провести експерименти над цією імплементацією та зробити висновки

Розділ 1. Опис алгоритму обчислення оберненої матриці

1.1. Означення

ОЗНАЧЕННЯ. Матриця A^{-1} називається оберненою для матриці $A = \| a_{ij} \|$, де $i = 1,2, \ldots, n$, а $j = 1,2, \ldots, n$, визначник якої не дорівнює нулю $| A | \neq 0$ та справджується рівність $A * A^{-1} = A^{-1} * A = E$, де E – одинична матриця порядку n на n.

ОЗНАЧЕННЯ. Означення оберненої матриці вводиться лише для квадратних матриць.

ОЗНАЧЕННЯ. Матриця A^T називається транспонованою, якщо виникає з матриці A в результаті операції транспонування: заміни рядків матриці на стовпчики.

ОЗНАЧЕННЯ. Мінор k-того порядку матриці A порядку m на n – визначник матриці порядку k на k, яка отримується із елементів матриці A, що знаходиться у визначених k стрічках і k стовбцях, при тому k менше за m та n.

ОЗНАЧЕННЯ. Алгебраїчним доповненням елементу a_{ij} квадратної матриці $A = \| a_{ij} \|$, де $i = 1,2, \ldots, n$, а $j = 1,2, \ldots, n$ називають мінор (n-1) порядку, який можна отримати із матриці A, викреслюючі елементи її i-тої стрічки й j-ого стовбця, помноженого на $(-1)^{i+j}$.

ОЗНАЧЕННЯ. Матриця $\tilde{A}^T$ називається приєднаною (союзною) до матриці A, якщо вона створена з алгебраїчних доповнень для відповідних елементів початкової матриці та транспонована по тому.

1.2. Знаходження оберненої матриці через приєднану

Зауважимо властивості визначника:

- $| A | = a_{p1} * A_{p1} + a_{p2} * A_{p2} + \ldots + a_{pn} * A_{pn} = a_{1q} * A_{1q} + a_{2q} * A_{2q} + \ldots + a_{nq} * A_{nq}$

  Де p = 1, 2, …, n, а q = 1, 2, …, n

- $a_{p1} * A_{q1} + a_{p2} * A_{q2} + \ldots + a_{pn} * A_{qn} = 0$
  $a_{1p} * A_{1q} + a_{2p} * A_{2q} + \ldots + a_{np} * A_{nq} = 0$

  Де p = 1, 2, …, n, а q = 1, 2, …, n, при тому $p \neq q$

Основуючись на цих властивостях, на означенні оберненої матриці та операції множення матриці на число можемо вивести рівність:

$$A^{-1} = \frac{1}{| A |} * \tilde{A}^{T}$$

Тобто, якщо приєднану матрицю поділити на визначник, то отримаємо обернену матрицю.

1.3. Обчислення приєднаної матриці

Алгоритм обчислення приєднаної матриці є наступним:

- Перевірити чи вхідна матриця є квадратною
- Обчислити визначник матриці A й перевірити, що він не дорівнює 0 (якщо він дорівнює 0, тоді матрицю не можна обернути)
- Побудувати матрицю з алгебраїчних доповнень відповідних елементів початкової матриці
- Матрицю, отриману на попередньому кроці, транспонувати, тим самим отримавши приєднану матрицю

1.4. Блочний алгоритм обчислення приєднаної матриці

Блочно-рекурсивний алгоритм заснований на вищеописаному алгоритмі, але в ньому присутні модифікації для ефективного розпаралелення та застосування рекурсії. Всього в алгоритмі 27 типів кроків для виконання

обчислення[8]. Їх можна розділити на 3 групи за специфікою. Першою є група кроків для обробки вхідних даних (розділення матриці на 4 рівні частини) та збирання з обчислених частин результату алгоритму. Другою є рекурсивні кроки цього алгоритму, застосованих на матрицях вдвічі меншої розмірності. Останньою групою є різноманітні обчислювальні кроки. Навантаження в алгоритмі розподілено таким чином, що кожен обчислювальний крок має лише одну «важку» операцію множення матриці на матрицю. Інші операції, як от: множення матриці на число, множення числа на число, сума матриць, рвзниця матриць – є відчутно менш часозатратні, тому їх не виділено в окремі кроки та використано в тих кроках, де одразу результати цих операцій буде використано або в тих кроках, де вхідні дані для цих операцій тільки що обчислено. В цій групі можна знайти підгрупи кроків, які ідентичні або схожі за своїми операціями, але мають різні вхідні дані. Виявлення цих груп допомогло скоротити кількість повторюваного коду до мінімуму в імплементації алгоритму, про що буде йти мова далі.

1.5. Граф блочного алгоритму обчислення приєднаної матриці

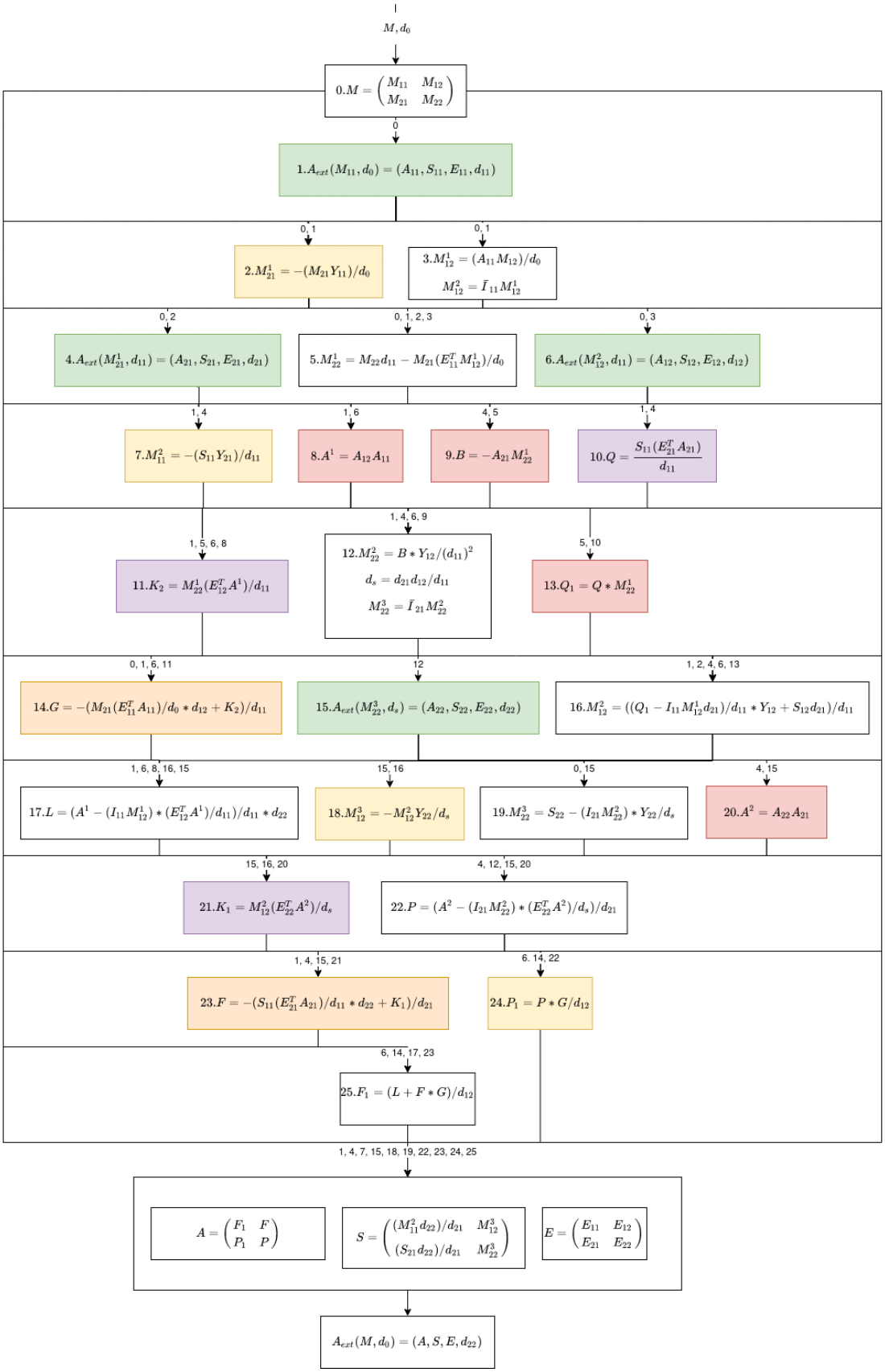

Рисунок 1. Граф блочного алгоритму обчислення приєднаної матриці

1.6. Блочний алгоритм множення матриць на 4 процесорах

Як було зазначено раніше, обчислення приєднаної матриці використовує в собі множення матриць, яке засноване на блочному алгоритмі, що може виконуватися паралельно на 4 процесорах.

Нехай дано матриці:

$$A = \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right),\ B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Де, a, b, c, d, e, f, g, h – блочні частини вхідних матриць

Задача: знайти C = A * B

Алгоритм обчислення:

- C_{11} = a*e + b*g
- C_{12} = a*f + b*h
- C_{21} = c*e + d*g
- C_{22} = c*f + d*h

$$C = \left(\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}\right)$$

Кроки незалежні один від одного, тому можуть паралельно виконуватися одночасно.

1.7. Граф блочного алгоритму множення матриць на 4 процесорах

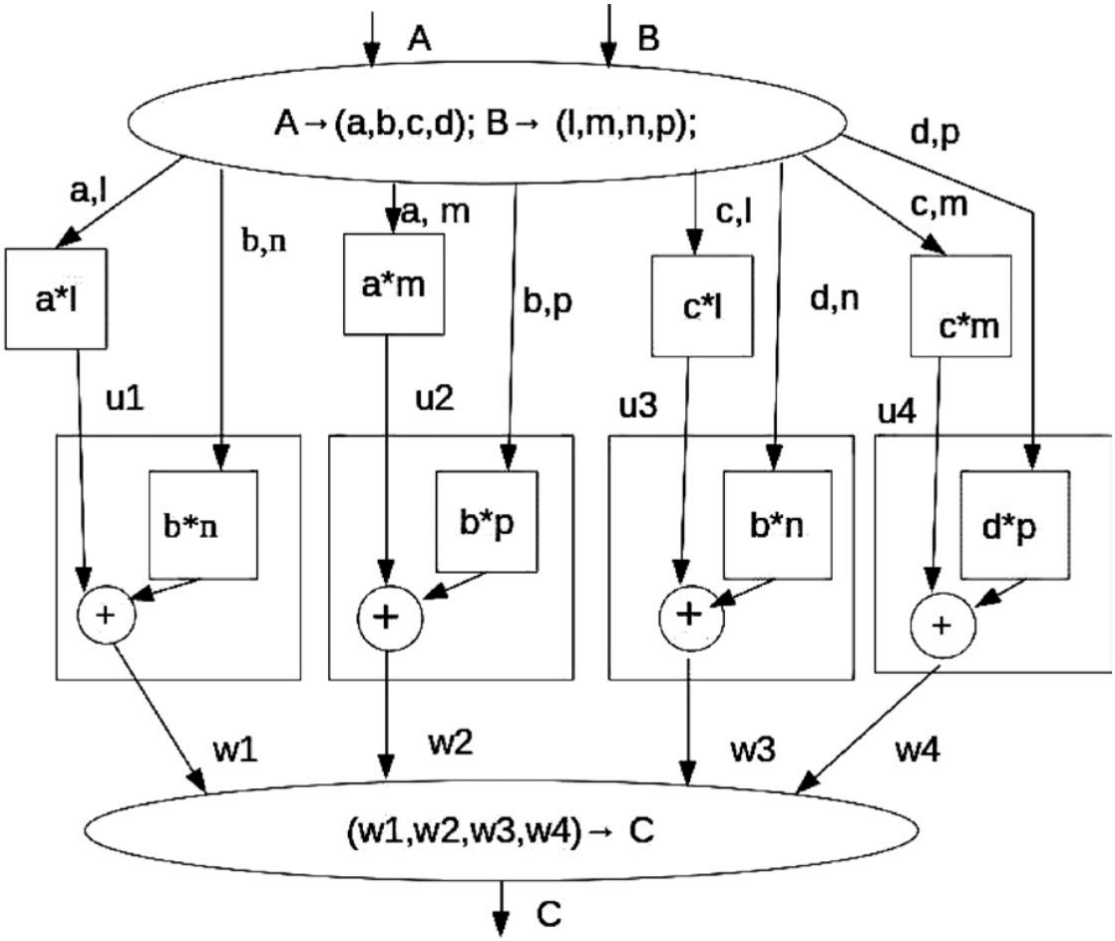

Рисунок 2. Граф блочного алгоритму множення матриць на 4 процесорах [5]

1.8. Реалізація за допомогою засобів стандартної бібліотеки Java

Було створено 3 версії обчислення, остання з яких – імплементація, що ґрунтується на блочному алгоритму обчислення приєднаної матриці. Реалізацію створено на базі допоміжних класів, що є частиною пакету dap репозиторію Math Partner.

Перша версія програми була реалізована наступним чином: було виділено групи алгоритмічних кроків, взаємопов’язані між собою. Цими групами є:

- (5) дії, що призводять до вирахування приєднаної матриці;
- (4) дії, що призводять до вирахування ешелонної форми;
- (3) дії, результати, яких потрібні попереднім двом групам;
- (2) дії, що вираховують Ei та Ej;
- (1) дії, без результати яких інші дії не можуть розпочати роботу

Дія №2 запускається паралельно після завершення дії №1, потім виконується дія №3, після чого паралельно запускаються дії №4 та 5. Слід зазначити, що в діях 2 та 5 певні розрахунки також виконуються паралельно та синхронізуються до завершення блоку дії.

Рекурсивні частини (їх було 4) знаходяться в дії №1, де матриця ділиться на 4 рівні частини та для кожної з них вираховується своя приєднана матриця.

Було проведено розпаралелення методів рекурсивного множення. Нижче надано схема розпаралелення:

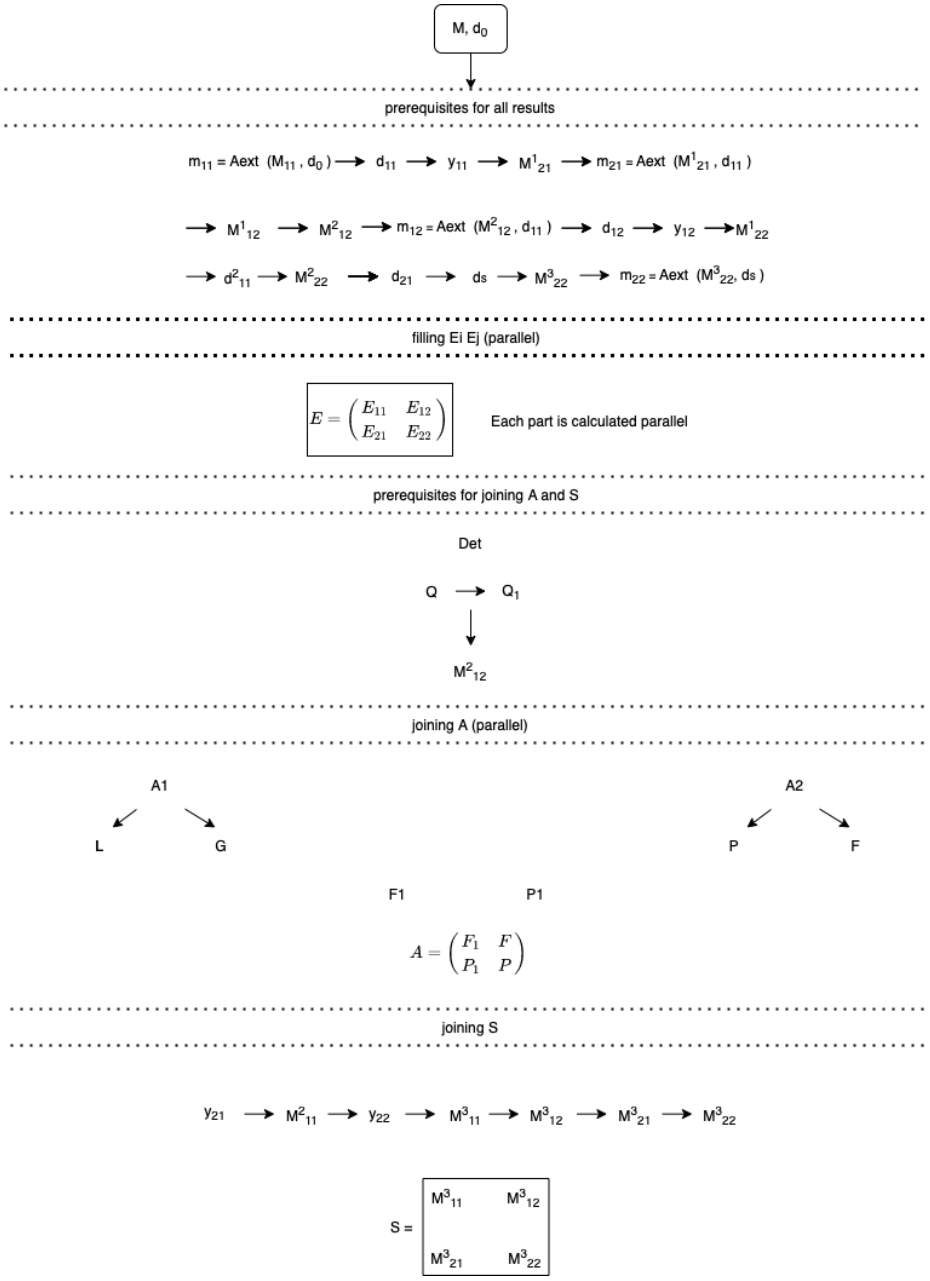

Рисунок 3 Схема розпаралелення за власною версією

Слід зазначити, що цей варіант розв’язку мав виграш у часі, але іноді закінчувалося виконання аварійно. Після досліджень проблеми було знайдено, що в рекурсивно-паралельних методах використовувалося в термінальній гілці метод множення, що не був придатним для паралельних обчислень. Спочатку було замінено алгоритм цього множення на простий алгоритм множення «стовпчик на рядок». Під час експериментів результати

показали, що на великих матрицях (від 2^8) паралельний варіант виконується більш ніж в два рази швидше.

Слідом був замінений алгоритм множення на інший високоефективний. Неочікувано сильно зменшився час виконання однопоточного варіанту. При розмірі матриці 2^{12} та виконанні термінальної гілки при розмірах 2^6 (ring.SMALLESTBLOCK) однопоточна програма закінчувала роботу в середньому через 12 секунд. Поекспериментувавши зі значенням ring.SMALLESTBLOCK в більшу сторону, вдалося при тих же розмірах матриці зменшити час виконання однопоточної програми до 2.5 секунд.

Було виведено, що більша рекурсивна глибина, яка регулюється значенням ring.SMALLESTBLOCK, тим менш ефективним є розпаралелення рекурсивного множення, оскільки багато часу та ресурсів витрачається на створення потоків та їхню синхронізацію. А оскільки імплементація алгоритму множення в термінальній гілці рекурсивного множення є високоефективною, то для будь-якого варіанту (як однопоточного, так й багатопоточного) збільшення значення ring.SMALLESTBLOCK до $2^{9\text{-}12}$ надає значну перевагу у часі.

Другий варіант розв’язку є покращенням першого через оптимізацію роботи із потоками, скорочення зайвої кількості, а також використання ExecutionService, а саме Executors.newFixedThreadPool. Для синхронізації виконання потоків використовувався метод submit, що повертає об’єкт типу Future, який надає можливість перевірити чи Runnable завершив свій процес виконання у ThreadPool. Як можна побачити на рисунку вище, результати виконання цього варіанту (підписано як average multi) виграють у часі за однопоточний варіант. Невеликий приріст у 9.8% пояснюється тим, що однопоточна програма й так високоефективна, а багатопоточний варіант має overhead через створення потоків та їхню синхронізацію. Перевірити

приріст на більшого розміру матриць не можна через OutOfMemoryError, що виникає при виклику на матрицях розміром 2^{13} й більше.

Третій варіант є імплементацією блочного алгоритму. Особливістю реалізації є розподіл виконання на 28 кроків (25 основних кроків та обчислення A, S, Ei та Ej), які виконуються паралельно та тісно пов'язані з результатом виконання попередніх кроків. Використовується ExecutionService, та для перевірки закінчення виконання попереднього кроку використовується метод isDone класу Future. У експериментах можна побачити швидке виконання, яке незначно відстає від попереднього багатопоточного варіанту. Це відставання пояснюється тим, що в цьому варіанті виконання більш розпаралелено, що створює більший overhead при його виконанні.

Нижче наведено фрагмент реалізації третього варіанту:

```
1.  // step 12 //
2.  AtomicReference<Element> d11_2 = new AtomicReference<>();
3.  AtomicReference<MatrixS> M22_2 = new AtomicReference<>();
4.  AtomicReference<Element> ds = new AtomicReference<>();
5.  AtomicReference<MatrixS> M22_3 = new AtomicReference<>();
6.  Future<?> step12 = service.submit(() -> {
7.      while( !(step4.isDone() && step6.isDone() && step9.isDone()) ) {}
8.      d11_2.set(d11.multiply(d11, ring));
9.          M22_2.set(B.get().multiplyDivRecursive(y12.get(),   d11_2.get().negate(ring),
    ring));
10.     ds.set(d12.get().multiply(d21.get(), ring).divide(d11, ring));
11.     M22_3.set(M22_2.get().multiplyLeftI(Array.involution(m21.get().Ei, finalN)));
12. });
13.
14. // step 13
15. AtomicReference<MatrixS> Q1 = new AtomicReference<>();
16. Future<?> step13 = service.submit(() -> {
17.     while( !(step5.isDone() && step10.isDone()) ) {}
18.     Q1.set(Q.get().multiplyRecursive(M22_1.get(), ring));
19. });
20.
21. // step 14
22. AtomicReference<MatrixS> G = new AtomicReference<>();
23. Future<?> step14 = service.submit(() -> {
24.     while( !(step6.isDone() && step11.isDone()) ) {}
25.     G.set((
26.                 M[2].multiplyDivMulRecursive(m11.A.multiplyLeftE(m11.Ej,  m11.Ei),  d0,
    d12.get(), ring).add(K2.get(), ring)
27.     ).divideByNumber(d11.negate(ring), ring));
28. });
29.
```

```
// step 15 //
AtomicReference<AdjMatrixS2> m22 = new AtomicReference<>();
AtomicReference<MatrixS> y22 = new AtomicReference<>();
Future<?> step15 = service.submit(() -> {
    while( !(step12.isDone()) ) {}
    m22.set(new AdjMatrixS2(M22_3.get(), ds.get(), ring));
    Det = m22.get().Det;
    y22.set(m22.get().S.ES_min_dI(Det, m22.get().Ei, m22.get().Ej, ring));
});

// step 16 //
AtomicReference<MatrixS> M12_2_new = new AtomicReference<>();
Future<?> step16 = service.submit(() -> {
    while( !(step4.isDone() && step6.isDone() && step13.isDone()) ) {}
    M12_2_new.set((
            (
                    (
                            Q1.get().subtract((M12_1.get().multiplyLeftI(m11.Ei).multiplyByNumber(d21.get(), ring)), ring)
                    )
                            .divideByNumber(d11, ring).multiplyRecursive(y12.get(), ring)
            )
                    .add((m12.get().S).multiplyByNumber(d21.get(), ring), ring)
    )
            .divideByNumber(d11, ring));
});
```

Для другого та третього варіантів було проведено додаткове розпаралелення рекурсивного множення. Для цього також було використано ExecutionService, та для перевірки закінчення виконання ¾ рекурсивних кроків методів використовується метод isDone класу Future. Було помічено, що розпаралелення рекурсивного множення дає приріст у порівнянні з варіантом без нього.

Надається порівняльна таблиця рекурсивного множення з та без розпаралелення:

2^N	**Без розпаралелення**	**З розпаралеленням**	**На скільки розпаралелений варіант швидший**
5	5.4ms	4.4ms	18,519%

6	19.1ms	16.9ms	11,518%
7	152.8ms	132.1ms	13,547%
8	934.4ms	829.8ms	11,194%
9	6702.1ms	6672.9ms	0,436%
10	53058.3ms	51814.2ms	2,345%
11	393873.7ms	385751.2ms	2,062%

Зменшення приросту можна пояснити тим, що глибина рекурсії з більшим розміром матриці становиться більшою, а реалізацією ExecutionService є newFixedThreadPool з кількістю 4 потоків. Якщо, наприклад, взяти реалізацію newCachedThreadPool, то виникне інша проблема: вхід в новий виток рекурсії частіший, ніж звільнення потока від обчислення, тому кількість потоків у пулі зросте й, при досить великих матрицях, програма намагатиметься створити більше потоків, ніж максимально можливо в джаві й програма завершиться аварійно.

Нижче приведена порівняльна таблиця з експериментами (раунди з 10) (з розпаралеленим рекурсивним множенням):

2^N	**Час виконання однопоточної програми**	**Час виконання багатопоточної програми (своє**	**Час виконання багатопоточної програми**	**На скільки багатопоточна програма 1 швидша за**	**На скільки багатопоточна програма 2 швидша за**

		розпаралелення)	**(розпаралелення блочне)**	**однопоточну**	**однопоточну**
5	2.2ms	4.8ms	7ms	-118.182 %	--218.182 %
6	6.3ms	9.0ms	9.8ms	-26.984 %	-41.270 %
7	10.8 ms	11.7ms	15.2 ms	-8.333 %	-40.741 %
8	26.1 ms	28.8 ms	30.5 ms	-10.345 %	-16.858 %
9	89.9 ms	87.0 ms	95.5 ms	3.226 %	-6.229 %
10	228.2 ms	197.0 ms	218.3 ms	13.672 %	4.338 %
11	867.5ms	604.8ms	630.0ms	30.282%	27.378%
12	2509.5ms	2276.2ms	2798.5ms	9.297%	-11.516%

Порівняльна таблиця з експериментами багатопоточної програми (блочна версія) з розпаралеленим множенням та без:

2^N	**Без розпаралелення**	**З розпаралеленням**
5	11.3ms	7ms
6	11ms	9.8ms
7	16.5ms	15.2 ms
8	29.3ms	30.5 ms

9	87.6ms	95.5 ms
10	226.9ms	218.3 ms
11	634.0ms	630.0ms
12	2928.9ms	2798.5ms

Порівняльна таблиця з експериментами багатопоточної програми (своя версія) з розпаралеленим множенням та без:

2^N	**Без розпаралелення**	**З розпаралеленням**
5	6.8ms	4.8ms
6	12.2ms	9.0ms
7	13.8ms	11.7ms
8	28.6ms	28.8 ms
9	93.6ms	87.0 ms
10	194.0ms	197.0 ms
11	627.7ms	604.8ms
12	2340.2ms	2276.2ms

З написання цих імплементацій можна зробити висновки, що паралелізація програми має часові переваги над однопоточною версією. Це твердження справджується на прикладах використання матриць великих розмірів. При розпаралеленні рекурсивних методів множення важливо використовувати не надто маленький показник початку виконання термінальної гілки. Також треба пам’ятати, що треба бути обережним при використанні потоків у

рекурсивних методах, бо, якщо рекурсивна глибина буде великою, то це коштуватиме помітних часових ресурсів. При великих розмірах матриц використання паралелізації є корисним, чим більше розмір матриці, тим приріст значніший.

Розділ 2. Програма динамічного управління завданнями DAP. Опис структури даних

Програма DAP ґрунтується на децентралізованому управлінні обчислювальним процесом, що імплементує певний рекурсивний алгоритм. Управління здійснюється над кластером процесорів. Кожен процесор може як проводити обчислення в певній зоні обчислювального графа, так й передавати частину обчислення зі своєї зони підлеглому вільному процесору обчислювального кластера[5].

Розгортання обчислювального процесу для обчислення рекурсивного алгоритму відбувається в декількох етапах. По-перше, будується обчислювальне дерево, що є розгорткою рекурсивних обчислень від кореневої (початкової) до листкових (кінцевих), до того ж вузли – рекурсивні функції. По-друге, проходить обчислення достатньо компактних даних на листках обчислювального дерева. У кінці повертаються значення результатів від листкових вершин обчислювального графу до попередніх витків рекурсії, аж до початкового виклику, в процесі з’єднуючи результати воєдино. Останній етап можна вважати завершеним, якщо обчислений результат для кореневої вершини.

DAP відрізняється від інших програм тим, що при розгортанні в глибину рекурсії здійснюється передача іншим процесорам частин обчислювального дерева зі списком підлеглих їм вільних процесорів. Звільнені від обчислення процесори динамічно перерозподіляються, таким чином утворюючи баланс у навантаженні кластеру. При тому зберігаються стани при будь-якому витку рекурсії, отже процесор має свободу перемикання з одного обчислювального доручення на інше. Варто сказати, що немає чіткої субординації для процесорів кластера: будь-який процесор

може володіти будь-якою вершиною обчислювального дерева й він має повне право управління процесом розрахунків в цій вершині.

Отже, процесори обчислювального кластера та вершині обчислювального дерева є два види об’єктів, над якими здійснюються операції й завданням програми є прив’язка процесорів до вершин дерева.

Далі буде показано ієрархію класів, що використано для роботи програми, а також буде описуватися їхня роль у цьому процесі.

2.1. Ієрархія класів

Далі подано ієрархію класів розробленої паралельної програми:

Package core:

- Amin
- CalcThread
- DispThread
- Drop
- Transport

Package multiply.MatrixS:

- MatrSMult4
- MatrSMultiplyScalar

Package adjmatrix.MatrixS:

- MatrSAdjMatrix

2.2. Діаграма класів

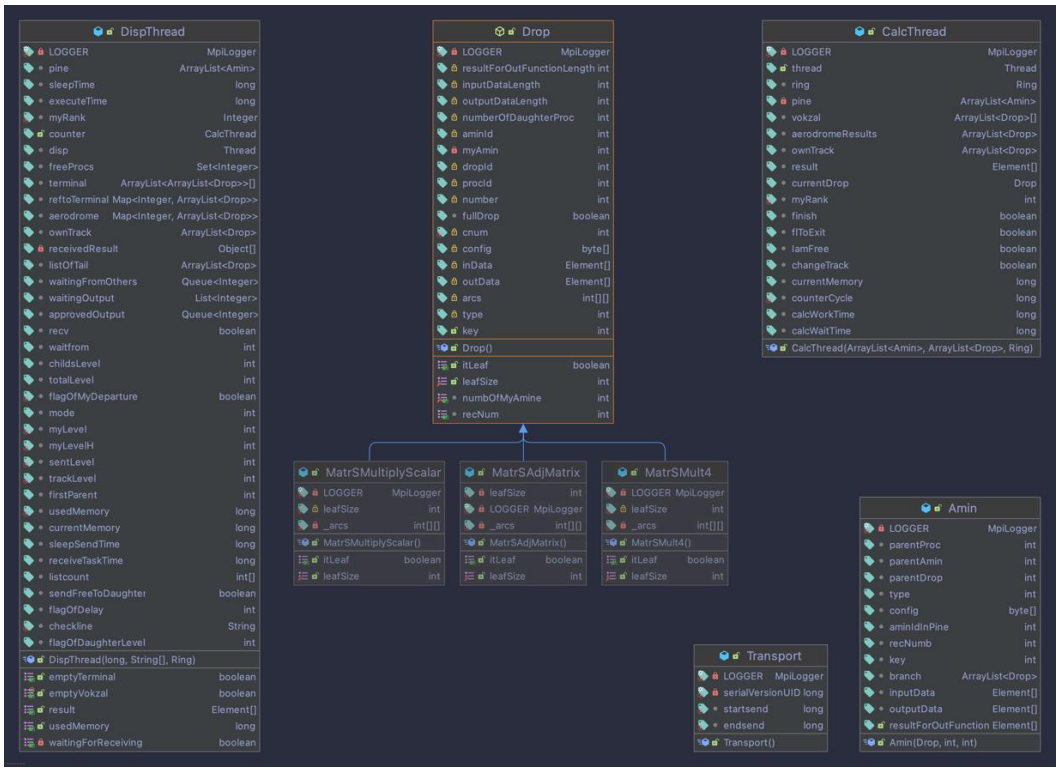

Рисунок 4. Діаграма класів

2.3. Дроп

Обчислювальне дерево розбиваються на окремі обчислювальні графи. Їхня назва – дропи. Входом для дропу є вершина обчислювального дерева, що обраховується рекурсивно (R-вершина). Оскільки передавання обчислювальних блоків є доволі часовитратною задачею, то такі вершини повинні мати нелінійну складність обчислення, щоб був сенс надавання можливості їхнього транспортування на розрахунок іншим процесорам.

Дроп – найменша складова частина обчислювального графа, що придатна для передачі іншим процесорам. До дропу може входити не лише однпа рекурсивна вершина, а ще й ті, у які можна потрапити з першої при умові, що вони недоступні через інші вершини. Обчислювальний граф є також зразком дропу. Для обчислення приєднаної матриці використовується дроп MatrSAdjMatrix, що має рекурсивні вершини себе та дропу MatrSMult4. У свою чергу MatrSMult4 має рекурсивні вершини MatrSMultiplyScalar, що в свою чергу мають рекурсивні вершини MatrSMult4.

Основні поля класу Drop

inData – масив вхідних даних, потріб - них для початку обрахунків. У кожного дропа свій розмір цього масиву, тому розмірність вказується індивідуально в кожному дочірньому класі. Доречно ініціалізовувати його в конструкторі, якщо невелика варіативність в обчисленні нерекурсивних частин дропу. Якщо ситуація не така й є багато різновидів (як буде далі розповідатися про MatrSMult4), то доречніше ініціалізувати в методі *setVars*.

outData – масив, що зберігає у собі результат обчислень дропу. Правила ініціалізації подібні до поля *inData*.

arcs – двовимірний масив, що зберігає у собі інформацію про те, якими є взаємозв'язки між частинами графу. Нульовим елементом цього масиву відповідає за вхідну функцію, а останній – за вихідну. Залежності одного дропу від іншого можна вказати за допомогою 3 чисел. Першим числом є номер дропу в графі, розпочинаючи з нуля. Другим є індекс елементу результативного масиву дропу, від якого залежать. Третім є індекс елементу вхідного масиву дропу, який залежить від іншого. Логічно, що нульовий елемент масиву *arcs* не може мати залежностей, оскільки він є першим у списку; а від останнього не може бути залежностей, оскільки після нього нема більше елементів.

type – унікальний ідентифікатор дропа. Кожний дроп має таке значення, що вказує на те, яке завдання виконується. Кожний дроп має унаслідувати абстрактний клас Drop та імплементувати усі методи, вказані у специфікації.

resultForOutFunctionLength – довжина масиву, що є вхідним параметром для вихідної функції дропу. Правила ініціалізації подібні до поля *inData*.

inputDataLength - довжина масиву inData. Ініціалізується до ініціалізації поля *inData*.

outputDataLength – довжина масиву outData. Ініціалізується до ініціалізації поля *outData*.

numberOfDaughterProc – номер дочірнбого процесора, куди дроп був надісланий на обробку. Це потрібно для того, щоб доправити дані, що потрібні для кінцевого обрахунку завдання дропу, але не обов'язково потрібні з самого початку. Значеннями можуть бути:

- -2: дроп ще не був доданий до списку задач на вирішення й не був надісланий до дочірнього процесора

- -1: він вже доданий до списку задач на вирішення, але ще не надісланий
- 0…n – номер процессора, до якого було надіслано дроп

aminId - ідентифікатор аміну, в якому знаходиться дроп

dropId – ідентифікатор дропу в аміні

myAmin - ідентифікатор аміну на пайні, який був розгорнутий з цього дропа

procId – ідентифікатор процесора, в якому було створено дроп та зберігається в його пайні

recNum – виток рекурсії

number - унікальний номер об’єкту дропу

key – логічний підтип дропу

fullDrop – прапорець того, що дроп отримав усі вхідні данні

Основні методи класу Drop

doAmin() – створює список дропів, які треба обчислити, щоб отримати результат для дропу, що викликає цей метод. Викликається під час розгортання завдання для певного аміна та записується в поле *branch* даного аміна.

sequentialCalc() – послідовне обчислення листкової вершини обчислювального дерева. Використовує *inData* та *outData.*

inputFunction() – вхідна функція дропу. Тут доречно вказати як обробити дані перед тим як відправити їх на рекурсивне обчислення. Виконується одноразово після розгортання аміна.

outputFunction() – вихідна функція дропу. Кінцевий етап обробки даних, що прийшли як результат рекурсивного обчислення з додатковими даними, якщо вони існують для дропу. Виконується одноразово, коли всі дані готові для створення кінцевого результату.

isItLeaf() – перевіряє чи дроп є листковою вершиною. Це робиться перевіркою розміру матриці, що може піти на потенціальне рекурсивне обчислення.

hasFullInputData() – для перевірки, що дроп має всі йому потрібні дані

doNewDrop() – фабрика для створення нових дропів.

2.4. Амін

Створений амін для розгортки підграфу відповідно до обчислювального дропу. Він складається з дропів.

Основні поля класу Amin

parentProc – номер процесора (поточного чи батьківського),який надіслав дроп на розгортку й до якого буде надіслано результат.

parentAmin – ідентифікатор батьківського аміну на пайні, з якого початково було передано дроп

parentDrop – ідентифікатор дропа, з якого даний амін розгорнуто. Батьківському процесору потрібно це поле, щоб знати в який дроп записати результат

type – тип дропу, з якого розгорнуто амін. Тип аміну та тип дропу співпадають.

aminIdInPine - ідентифікатор аміна в пайні

recNumb - виток рекурсії

key – логічний підтип дропу, з якого розгорнуто амін. Ключ аміну відповідає ключу дропу.

branch – список дропів, потрібних для обчислення поточного завдання

inputData – масив вхідних даних для поточного завдання. inputData аміну та inData дропу співпадають

outputData - масив вихідних даних для поточного завдання. outputData аміну та outData дропу співпадають

resultForOutFunction – масив, що зберігає проміжні розв’язки (обчислення дропів з *branch*) для використання у вихідній функції поточного дропу.

Основні методи класу Amin

конструктор класу – приймає такі параметри: власне дроп, з якого виконується розгортка,індекс аміна в пайні, номер батьківського процесора. Сам процес розгортки працює завдяки методу класу Drop *doAmin()* та проставляються мета-поля дропів, що зберігаються в полі *branch*.

setIndexToDrops() - проставляються мета-поля дропів, що зберігаються в полі *branch*. А саме такі поля: *aminId, dropId, procId, recNum*. Також для всіх дропів викликається метод *setVars()*.

hasFullOutput() – метод для перевірки, що вхідний масив для вихідної функції повністю наповнений.

2.5. Транспорт

Утілітний клас для інкапсуляції усіх методів, пов'язаних з передачею генералізованих даних за допомогою бібліотеки передачі повідомлень OpenMPI[9].

Основні методи класу Transport

probeAny() – неблокуюча перевірка того, що до поточного процесора було надіслано повідомлення, при тому перевіряється наявність всіх видів кастомних повідомлень та від будь якого ресурсу (процесора).

Існують такі види кастомних повідомлень:

- TASK (0) – повідомлення із завданням
- FREE_PROC (1) – повідомлення про звільнені процесори
- PROC_STATE (2) – повідомлення про стан процесора
- RESULT (3) – повідомлення, що містить результат обрахунків
- FINAL (4) – сигнал закінчення операцій
- REQUEST_TO_APPROVE (5) - прохання надати згоду на надіслання блокувального повідомлення
- APPROVAL (6) – згода на повідомлення (5)
- CANCEL (7) – відмова на повідомлення (5)

iSendIntArray() - надіслання масиву цілих чисел, неблокуюча операція

receiveIntArray() – отримання масиву цілих чисел, блокуюча операція

sendObject() - надіслання серіалізованого об'єкту, блокуюча операція

sendObjects() - надіслання серіалізованого масиву об'єктів, блокуюча операція

recvObject() – отримання серіалізованого об’єкту, блокуюча операція

recvObjects() - отримання серіалізованого масиву об’єктів, блокуюча операція

2.6. Пайн

Аміни розгортаються з дропів, а всі аміни, що належать одному процесору зберігаються в пайні. Тобто пайн – реалізація обліку амінів на кожному з процесорів.

Оскільки обчислення дропів можуть передаватися дочірним процесорам, то потрібно мати адресу, на яку треба повернути обчислений результат. Ця адреса (PAD) формується з 3 ідентифікаторів: ідентифікатор процесора (np), ідентифікатор аміна в пайні (na), ідентифікатор дропа в аміні (nd). При передачі обчислення в дочірній процесор, на ньому створиться відповідний даному амін та буде розміщений в локальному пайн.

Комунікацію між процесорами підтримується завдяки таким об’єктам, як: Вокзал, Аеродром, Термінал

2.7. Вокзал

Вокзал – список дропів, що чекають початку свого обчислення. Вокзал має декілька рівнів, на яких розташовуються різні завдання в залежності від того який вони мають рівень (виток) рекурсії. Пріоритетнішими для обробки є дропи з найнижчим рівнем рекурсії. Найнижчий рівенб рекурсії є поточним рівнем вокзалу.

Вокзал поповнюється таким чином: кожен раз, коли виявляється при розгортці аміна з нового дропу, що піддропи можна обчислити паралельно, поповнюється вокзал цими піддропами.

Поки немає ще ієрархічних (батьківських) зв’язків між процесорами, є вільні процесори, а також вокзал має завдання, дропи найнижчого рівня

розподіляються на кількість вільних процесорів. Підлеглі поточному процесори можуть бути отримані від батьківського процесора або від самого дочірнього процесора завдяки передачі відповідного повідомлення. Процесор вважається вільним, якщо всі дропи йому надані обчислені та його вокзал стає порожнім.

2.8. Аеродром

Аеродром – об’єкт типу Map, що зберігає батьківські процесори разом із дропами, які ці батьківські процесори видали своїм дочірнім. Саме факт видачі завдання іншому процесору робить його батьківським. Дроп видаляється зі списку батьківського процесора в аеродромі, коли результати цього дропу обчислені та передані батьківському процесору. Процесор видаляється з аеродрому, коли всі дропи повернули йому результати своїх обчислень.

2.9. Термінал

Термінал слугує комунікатором з дочірніми процесами. В терміналі вони зберігаються подібно до вокзалу – по рівням. Й так само рівень терміналу вважається найнижчий рівень з дочірніх процесорів.

Рівень процесора обраховується як найменше число серед рівня вокзалу та рівня терміналу. При зміні рівня процесора, процесор відправляє своїм батьківським процесорам повідомлення про цю зміну.

Розділ 3. Різновиди потоків

У системі використовуються два види потоків, які запущені на кожному процесорі по одному екземпляру: потік-диспетчер та обчислювальний потік. Обчислювальний потік розраховує обчислення за інструкціями, що відповідають поданому алгоритму; диспетчер керує обробкою (надісланням і прийманням) повідомлень від інших процесорів та управлінням обчислень.

3.1. Потік-диспетчер

Реалізований за допомогою нескінченного циклу, задача цього потоку – управління повідомленнями та підлеглими процесорами. Цикл може закінчитися в той момент, коли сигнал про закінчення обрахунків всього вхідного алгоритму буде подано. Водночас працює один потік: або диспетчерський, або обчислювальний. Коли диспетчер прокидається, перевіряється чи було надіслано повідомлень від інших процесорів та оброблюються ці повідомлення. Різновиди цих повідомлень було описано в розділі 2.5.

Такими є різновиди завдань, що виконує потік-диспетчер:

- Прийом завдання
- При наявності необчислених дропів та вільних процесорів, відправлення їм таких дропів
- Прийом підлеглих процесорів
- Перенаправлення підлиглих процесорів дочірнім
- Обробка стану дочірних процесорів
- Передача себе та своїх підлеглих процесорів під підпорядкуванням неопосередкованому батьківському процесору
- Прийом результату обрахунків дропа

- Відправлення результатів батьківським процесорам
- Очікування сигналу завершення

Основні методи класу DispThread

pine – список амінів

myRank – номер процесору

counter – посилання на обчислювальний потік

disp – посилання на потік диспетчер (на самого себе)

freeProcs – список вільних процесорів

terminal - імплементація терміналу. Імплементовано списком списків дропів, що були надіслані дочірнім процесорам за конкретним індексом. Позиція в списку відповідає за рівень дочірнього процесора.

refToTerminal – мапа створена для швидково доступу за ключем (ідентифікатором дочірнього процесора) до посилання дропа в терміналі

aerodrome - мапа, що імплементує аеродром з ідентифікатором батьківського процесора як ключ та значення у вигляді списку дропів, що надійшов від конкретного батьківського процесора. При порожньому списку, ключ батьківського процесора видаляється з мапи.

ownTrack - список дропів, що залишаються на обчисленні в даному процесорі. В ньому зберігаються нерекурсивні дропи, дропи-результати.

result – масив результатів завдання

receivedResult – масив результатів, отриманих від дочірнього процесора

waitingFromOthers – черга тих процесорів, що просять надіслати блокувальне повідомлення

waitingOutput – список процесорів, кому надіслано запит на відправлення блокувального повідомлення

approvedOutput – черга процесорів, що підтвердили згоду на отримання

recv – прапорець чергування відправлення та прийому блокувального повідомлення. True для прийому, False (значення за замовченням) для відправлення

waitfrom – ідентифікатор процесору, від якого очікується блокуюче повідомлення

childsLevel – рівень дочірніх процесорів – найменший індекс терміналу з непорожнім списком

totalLevel – рівень процесора: мінімальне між *myLevel* та *childsLevel*

flagOfMyDeparture – прапорець, що показує, що даному процесору можна надати нове завдання. Працює воно наступним чином: процесор самостійно себе записує в список вільних процесорів та відправляє або батьківському процесору або дочірньому

myLevel – найнижчий рівень вокзалу поточного процесора

myLevelH – найвищий рівень вокзалу поточного процесора

sentLevel – останній надісланий до батьківського процесора рівень рекурсії

trackLevel – найнижчий рівень *ownTrack*

firstParent – ідентифікатор безпосереднього батьківського процесора

Основні методи класу DispThread

rootWork() – метод кореневого процесора, що виконується перед запуском диспетчера. Записуються до нього всі процесору у ролі вільних та створюється початковий дроп та заноситься у *ownTrack*.

exit() – подання сигналу про завершення програми

receiveTask() – отримання нового дропу від батьківського процесора. Якщо це було першим завданням надісланим від батька, то заповнюється

firstParent батьківським ідентифікатором. Батько видаляється з черги *waitingFromOthers, recv* ставиться false. *flagOfMyDeparture* ставиться false.

receiveFreeProcs() – отримання списку звільнених процесорів та додавання до своїх

procLevel() – отримання стану дочірнього процесора, записання на новий рівень терміналу його дропів та видалення списку зі старої позиції. Відбувається зміна рівня вокзалу, якщо новий рвіень менший за нього.

endProgramme() – метод, що викликається кореневим процесором для сигналювання закінчення програми всім процесорам та зупинки свого циклу диспетчера.

receiveResult() – отримує результат дропу від дочірнього процесора та записує результат в оригінальний дроп на своєму пайні та кидає на свій трек *ownTrack*. Видаляє дроп зі списку дочірнього або зразу, або наступного разу в разі отриманні нового результату через оновлення дочірнього процесора свого нового рівня. Батько видаляється з черги *waitingFromOthers, recv* ставиться false.

addDaughter() – додавання дропу до списку дочірнього процесора у термінал, якщо процесор присутній. Якщо ні, то створення нового списку та додавання його в 20-тий останній рівень терміналу. Оновлення *refToTerminal.*

deleteDaughter() – видалення дропу зі списку дочірнього процесора в терміналі. При відсутності інших дропів у списку, видаляється дочірній процесор з терміналу та *refToTerminal*. При відсутності інших дочірніх процесорів на рівні терміналу, шукається інший непорожній рівень та встановлюється новий *childsLevel*. При пустому терміналі ставиться рівню терміналу значення 20.

deleteParent() – видалення обрахованого дропу батьківського процесора з аеродрому. При відсутності інших дропів, батьківський процесор видаляється з мапи.

addParent() – додавання нового дропу батьківського процесора в аеродром. При відсутності батьківського процесора додається ключ до мапи.

isEmptyVokzal() – перевірка одночасного виконання 2 умов: *myLevel == 20 && myLevelH == 20.*

sendDrops() – відправлення вільного дропу дочірньому процесору, що погодився на отримання, та надсилання вільних процесорів. При пустому вокзалі повертається значення false.

sendDrop() – процес відправлення вільного дропу дочірньому процесору, видалення цього процесору зі списку вільних.

sendRequestToSendDrops() – відправлення запиту на згоду отримати новий дроп дочірнім процесорам. При цьому процесор видаляється зі списку вільних процесорів.

sendRequestToApproveSending() – надсилання запиту на отримання блокувального повідомлення.

receiveRequestToApproveReceiving() – отримання запиту на отримання блокувального повідомлення.

approveReceiving() – відправлення підтвердження отримання блокувального повідомлення.

cancelSending() – скасування відправлення дропа, додавання процесору відправлення до списку вільних процесорів.

receiveApprovalForSending() – отримання підтвердження на запит про відправлення блокувального повідомлення, додавання процесора в список *approvedOutput.*

receiveCancel() – отримання скасування відправлення повідомлення, зміна прапорцю *recv* на false, перехід у стан відправлення повідомлень.

isWaitingForReceiving() – перевірка, чи очікується отримання повідомлення.

sendFreeProc() – надсилання дочірньому процесору вільних процесорів. Кількість залежить від розміру вокзалу: якщо 0, то надсилає всіх вільних, інакше кількість вільних поділена на розмір вокзалу плюс 2.

sendFreeToDaughter() – розсилання вільних процесорів дочірнім, що знаходяться в терміналі на найнижчому рівні, допоки список вільних процесорів не буде порожнім.

isEmptyTerminal() – перевіряється чи термінал пустий

sendFreeProcs() – надсилання вільних процесорів, включаючи з собою, до батьківського або дочірніх процесорів; якщо порожній вокзал, наявні дочірні процесори із завданням та рівні поточного процесора й дочірнього (*childsLevel*) більше двох,то надсилається дочірнім. Якщо ж це не кореневий процесор, рівень терміналу став дорівнювати 20 та вокзал порожній, тоді надсилається батьківському.

doMeFree() – додавання свого процесора до списку вільних та зміна *flagOfMyDeparture* на true.

sendLevel() – зміна рівня процесора та надсилання батьківським процесорам, які зберігаються в аеродромі, та оновлюється зміна *sentLevel.*

tagAction() – перенаправник повідомлень за їхнім типом

sendResultsToParent() – надсилання результату дропу батьківському процесору, який цього запитував.

sendRequestsForResultsSending() – надсилання запиту на відправлення результату обрахування дропу.

resetFields() – очистка та повернення до початкових значень всіх полів класу.

clear() – очистка списків, мап.

execute() - старт програми, де запускається цикл диспетчерського потоку, якщо це нульовий поток, то й виконує метод *rootWork()*. Всередині реалізований функціональний цикл, в якому як раз перевіряється чи надійшли нові повідомлення, оброблюються ці повідомлення, створює сам повідомлення. Може виконувати в циклі легку роботу або, якщо не очікуються блокувальні повідомлення, то робить важку роботу та результати відправляє. У кінці ітерації циклу засинає для пріоритизації роботи обчислювального потоку.

doLiteJob() – виконання легких завдань: при непустому аеродромі виклик методу *sendLevel()* або при непустому списку вільних процесорів виклик методу *sendFreeProcs()*.

makeRequestsForSending() – при можливості розсилка необчислених дропів або відправлення отриманих результатів обрахунків.

doHardWork() – надсилання блокувальних повідомлень процесорам, що погодилися їх прийняти. Спочатку виконується спроба відправлення результату батьківським процесорам, якщо не вдалося, то надсилаються дропи, якщо й це не вдалося, то скасовується відправлення.

3.2. Обчислювальний потік

Очікує надходження дропу та запускає обчислення.

Основні поля класу CalcThread

thread - посилання на даний потік

ring - математичне кільце, в якому проводяться обчислення.

pine – посилання на пайн, що є спільним з потоком-диспетчером поточного процесора

vokzal – посилання на вокзал, що є спільним з потоком-диспетчером поточного процесора. Обчислювальний потік бере завдання з поверхневого рівня (*myLevelH*) вокзалу, де зберігаються дропи з найбільшою глибиною рекурсії.

aerodromeResults – список готових результатів, що очікують відправлення до батьківського процесора.

ownTrack – посилання на список дропів для поточного процесора

result – масив з результатами початкового дропу.

currentDrop - розгорнутий на даний момент дроп на поточному процесорі

myRank – номер поточного процесора

finish – прапорець, що вказує на закінчення обчислень

flToExit - прапорець, що сигналізує завершення роботи обчислювального потоку

IamFree – прапорець, що вказує на порожній стан вокзалу та свого треку й можливість обчислювального потоку отримати нове завдання

Основні методи класу CalcThread

DoneThread() – зупинка обчислювального потоку

putDropInTrack() – додавання дропу до списку свого треку, якщо глибина рекурсії менша, ніж *trackLevel* диспетчера, тоді оновлюємо це поле значення глибини рекурсії дропу.

putDropInVokzal() - додавання дропу до вокзалу, після цього йде процесс оновлення значень верхньої та нижньої межі (рівня) вокзалу.

writeResultsToAmin() – передача результатів дропа в інші відповідно до залежностей, вказаних через поле *arcs* дропу. У цьому ж методі викликаються методи *putDropInVokzal(), putResultsToAminOutput().*

writeResultsAfterInpFunc() – передача вихідних даних вхідної функції дропу у вхідні дані дропів аміну відповідно до залежностей, вказаних через поле *arcs* дропу. У цьому ж методі викликається метод *putDropInVokzal()* при заповненні всіх вхідних даних певного дропу.

addToAerodromeResults() – результати обчислення дропу додаються до списку aerodromeResults.

putResultsToAminOutput() – виконання вихідної функції аміну. Якщо це кореневий дроп – то закінчення роботи програми. Якщо дроп був переданий батьківським процесором, то виконується виклик методу *addToAerodromeResults().* Інакше - результат записується в дропі, який належить батьківському даному аміну на тому ж пайні.

finishWholeTask() – подання сигналу про закінчення всіх обчислень.

getTask() – взяття нового дропу або обчислювальним потоком або потоком-диспетчером. Обчислювальний бере дроп зі свого треку або з вокзалу, якщо попередній порожній. Якщо це був останній дроп рівня вокзалу, то викликається метод *changeMyLevelH().* Диспетчер бере дроп з нижнього рівня й при необхідності викликає метод *changeMyLevel().*

changeMyLevelH() – зміна верхнього рівня вокзалу.

changeMyLevel() – зміна нижнього рівня вокзалу.

changeTrackLevel() – зміна рівня своґго треку.

deleteFromTrack() – видалення дропу зі свого треку.

inputDataToAmin() – обчислення вхідної функції дропу та виклику методу *writeResultsAfterInpFunc().* У цьому ж методі ж викликається метод дропу *independentCalc().*

run() – метод класу, де імплементовано нескінченний цикл, який завершується при зміні прапорця *flToExit.* Якщо вокзал та трек непорожні, то виконується метод *ProcFunc()*, інакше прапорець *IamFree* спсравджується та очікується надходження нових дропів.

ProcFunc() – оброблює дроп, взятий або з вокзалу або зі свого треку. Якщо дроп вже має вихідні дані, то викликається метод

writeResultsToAmin(). Інакше перевіряється чи є листковим дроп. Якщо він не є листковим, то розгортається новий амін викликом методу *inputDataToAmin().* Інакше виконується послідовний обрахунок дропу. Якщо це кореневий процесор, то викликається метод *finishWholeTask().* Якщо це дроп поточного процесору, то викликається метод *writeResultsToAmin(),* інакше викликається метод *addToAerodromeResults*().

Розділ 4. Дропи для обчислення приєднаної матриці

Для створення дропу потрібно успадкувати абстрактний клас *Drop* та перевантажити всі його абстрактні методи й при необхідності ще додаткові, про що буде йти мова далі. У дропі обчислення приєднаної матриці *MatrSAdjMatrix* в амін додаються рекурсивні дропи та дропи обчислення множення матриць на 4 процесорах *MatrSMult4*. У свою чергу в амін *MatrSMult4* додаються дропи *MatrSMultiplyScalar*. А в їхніх амінах додаються дропи *MatrSMult4*. Треба зазначити, що використано для обчислень екземпляри класів з їхніми методами з бібліотеки MathPartner. Розпочнемо з розгляду класу *MatrSMultiplyScalar*, оскільки його імплементація найпростіша.

4.1. Клас MatrSMultiplyScalar

Для того щоб визначити межу, коли дроп можна вирахувати послідовним чином, використовується поле *leafSize* із сеттером *setLeafSize*. Сама перевірка виконується методом *isItLeaf()*, що в даному класі реалізовано перевіркою розмірності першого аргументу вхідних параметрів дропу.

```
1.  @Override
2.  public boolean isItLeaf() {
3.      MatrixS ms = (MatrixS) inData[0];
4.      return (ms.size <= leafSize);
5.  }
```

Для декларування залежностей вхідних параметрів одного дропу від вихідних параметрів іншого використовується поле *arcs*.

```
1.  private static int[][] _arcs = new int[][]{
2.      { // 0ий дроп (inputFunction)
3.          1, 0, 0, // перший дроп приймає 0ий вихідний аргумент як свій 0ий вхідний
    аргумент
4.          1, 1, 1, // перший дроп приймає 1ий вихідний аргумент як свій 1ий вхідний
    аргумент
5.          2, 2, 0, // другий дроп приймає 2ий вихідний аргумент як свій 0ий вхідний
    аргумент
6.          2, 3, 1  // другий дроп приймає 3ий вихідний аргумент як свій 1ий вхідний
    аргумент
7.      },
8.      {3, 0, 0}, // 3ий дроп: третій дроп приймає 0ий вихідний аргумент як свій 0ий
    вхідний аргумент
```

```
9.      {3, 0, 1}, // 3ий дроп: третій дроп приймає 0ий вихідний аргумент як свій 1ий
    вхідний аргумент
10.     {} // 3ій дроп (outputFunction)
11. };
```

Як зазначалося раніше, ініціалізацію полів, що вказують на тип дропу, розмір масивів *inData*, *outData*, *arcs* та *number* вказано для простоти в конструкторі:

```
1.  public MatrSMultiplyScalar() {
2.      resultForOutFunctionLength = 2;
3.      inputDataLength = 4;
4.      inData =  new Element[4];
5.      outData =  new Element[1];
6.      type = 7;
7.      number = cnum++;
8.      arcs = _arcs;
9.  }
```

Наступним чином декларовано метод *doAmin()*:

```
1.  @Override
2.  public ArrayList<Drop> doAmin() {
3.      ArrayList<Drop> amin = new ArrayList<Drop>();
4.      amin.add(new MatrSMult4());
5.      amin.add(new MatrSMult4());
6.      return amin;
7.  }
```

Послідовне множення матриці на скаляр виражено в коді так:

```
1.  @Override
2.  public void sequentialCalc(Ring ring) {
3.      MatrixS A = (MatrixS) inData[0];
4.      MatrixS B = (MatrixS) inData[1];
5.      MatrixS C = (MatrixS) inData[2];
6.      MatrixS D = (MatrixS) inData[3];
7.      MatrixS R = A.multiply(B, ring).add(C.multiply(D, ring), ring);
8.      outData[0] = R;
9.  }
```

Вхідна функція обробляє надіслані до дропу дані та розбиває на блоки:

```
1.  @Override
2.  public MatrixS[] inputFunction(Element[] input, Amin amin, Ring ring) {
3.      MatrixS[] res = {(MatrixS) input[0], (MatrixS) input[1], (MatrixS) input[2],
    (MatrixS) input[3]};
4.      return res;
5.  }
```

Вихідна функція збирає блоки-результати з рекурсивних дропів розгорнутого аміну та формує результат:

```
1.  @Override
```

```
2.  public MatrixS[] outputFunction(Element[] input, Ring ring) {
3.      MatrixS A = (MatrixS) input[0];
4.      MatrixS B = (MatrixS) input[1];
5.      MatrixS[] res = new MatrixS[]{A.add(B, ring)};
6.      return res;
7.  }
```

4.2. Клас MatrSMult4

Якщо розглянути рисунок 1, то можна побачити, що 21 крок зосереджені навколо множення двох матриць. Щоб не створювати велику кількість класів, які будуть унаслідувати клас *Drop* й при тому сильно повторюватися, було створено поле *key* у класі *Drop*. Воно вказує, яку саме варіацію дропу треба обраховувати. За замовченням, значення поля *key* дорівнює 0. Для зручності обліку цих ключів, ключи варіацій дропів, які будуть використовуватися при обчисленні приєднаної матриці, чотирьохзначні та починаються на 77 (M у таблиці ASCII, початкова літера прізвища автора). Останні дві цифри відповідають номеру крока в блочному алгоритмі, відображеному на рисунку 1.

Нижче наведено поле *arcs* класу *MatrSMult4*, логіка побудови залежностей відповідає логіці, наведеної в прикладі з попереднім класом:

```
1.  private static int[][] _arcs = new int[][]{
2.      { // 0ий дроп (inputFunction)
3.          1, 0, 0,   1, 4, 1,   1, 1, 2,   1, 6, 3,
4.          2, 0, 0,   2, 5, 1,   2, 1, 2,   2, 7, 3,
5.          3, 2, 0,   3, 4, 1,   3, 3, 2,   3, 6, 3,
6.          4, 2, 0,   4, 5, 1,   4, 3, 2,   4, 7, 3
7.      },
8.      {5, 0, 0}, // 1ий дроп
9.      {5, 0, 1}, // 2ий дроп
10.     {5, 0, 2}, // 3ій дроп
11.     {5, 0, 3}, // 4ий дроп
12.     {} // 5ий дроп (outputFunction)
13. };
```

Оскільки варіації дропа, основною операцією якої є множення, відрізняються різними значеннями полів *inputDataLength*, *outputDataLength*, *resultForOutFunctionLength*, то краще винести їхнє присвоєння у метод *setVars()*:

```
1.  @Override
2.  public void setVars(){
3.      switch (key){
4.          case(7708):
5.          case(7709):
6.          case(7713):
7.          case(7720): {
8.              inputDataLength = 2;
9.              outputDataLength = 1;
10.             resultForOutFunctionLength = 4;
11.             break;
```

```
12.         }
13.         case(7702):
14.         case(7707):
15.         case(7710):
16.         case(7718):
17.         case(7724): {
18.             inputDataLength = 3;
19.             outputDataLength = 1;
20.             resultForOutFunctionLength = 4;
21.             break;
22.         }
23.         case(7703): {
24.             inputDataLength = 4;
25.             outputDataLength = 2;
26.             resultForOutFunctionLength = 4;
27.             break;
28.         }
29.         case(7705): {
30.             inputDataLength = 6;
31.             outputDataLength = 1;
32.             resultForOutFunctionLength = 5;
33.             break;
34.         }
35.         case(7711):
36.         case(7721):
37.         case(7725): {
38.             inputDataLength = 4;
39.             outputDataLength = 1;
40.             resultForOutFunctionLength = 4;
41.             break;
42.         }
43.         case(7712): {
44.             inputDataLength = 7;
45.             outputDataLength = 3;
46.             resultForOutFunctionLength = 5;
47.             break;
48.         }
49.         case(7714):
50.         case(7717):
51.         case(7722):
52.         case(7723): {
53.             inputDataLength = 6;
54.             outputDataLength = 1;
55.             resultForOutFunctionLength = 4;
56.             break;
57.         }
58.         case(7716): {
59.             inputDataLength = 7;
60.             outputDataLength = 1;
61.             resultForOutFunctionLength = 5;
62.             break;
63.         }
64.         case(7719): {
65.             inputDataLength = 5;
66.             outputDataLength = 1;
67.             resultForOutFunctionLength = 4;
68.             break;
69.         }
70.     }
71.     inData  = new Element[inputDataLength];
72.     outData = new Element[outputDataLength];
73. }
```

У контексті цього метода багато кейсів можна згрупувати в одну підгрупу, як показано вище. Значення, що однакові для всіх варіацій дропа, ініційовано у конструкторі:

```
1.  public MatrSMult4() {
2.      type = 5;
3.      number = cnum++;
4.      arcs = _arcs;
5.  }
```

Наступним чином декларовано метод *doAmin()*:

```
1.  @Override
2.  public ArrayList<Drop> doAmin() {
3.      ArrayList<Drop> amin = new ArrayList<Drop>();
4.      amin.add(new MatrSMultiplyScalar());
5.      amin.add(new MatrSMultiplyScalar());
6.      amin.add(new MatrSMultiplyScalar());
7.      amin.add(new MatrSMultiplyScalar());
8.      return amin;
9.  }
```

Виконання методу *sequentialCalc()* напряму залежить від номеру варіації дропу:

```
1.  @Override
2.  public void sequentialCalc(Ring ring) {
3.      switch (key){
4.          case(7702):
5.          case(7718): {
6.              MatrixS A = (MatrixS) inData[0];
7.              MatrixS B = (MatrixS) inData[1];
8.              Element d = inData[2];
9.              outData[0] = A.multiplyDivRecursive
10.                 (B, d.negate(ring), ring);
11.             break;
12.         }
13.         // кейси 7703, 7705, 7707
14.         // було пропущено для компактності прикладу
15.         case(7708):
16.         case(7720): {
17.             MatrixS A1 = ((AdjMatrixS) inData[0]).A;
18.             MatrixS A2 = ((AdjMatrixS) inData[1]).A;
19.             outData[0] = A1.multiply(A2, ring);
20.             break;
21.         }
22.         // кейси 7709, 7710, 7713
23.         // було пропущено для компактності прикладу
24.         case(7711):
25.         case(7721): {
26.             MatrixS M22_1 = (MatrixS) inData[0];
27.             MatrixS A1 = (MatrixS) inData[1];
28.             AdjMatrixS m12 = (AdjMatrixS) inData[2];
29.             Element d11 = inData[3];
30.             outData[0] = M22_1.multiplyDivRecursive
31.                 (A1.multiplyLeftE(m12.Ej, m12.Ei), d11, ring);
32.             break;
33.         }
34.         case(7712): {
```

```
35.             MatrixS B = (MatrixS) inData[0];
36.             MatrixS y12 = (MatrixS) inData[1];
37.             Element d11 = inData[2];
38.             Element d21 = inData[3];
39.             Element d12 = inData[4];
40.             AdjMatrixS m21 = (AdjMatrixS) inData[5];
41.             Element finalN = inData[6];
42.             Element d11_2 = d11.multiply(d11, ring);
43.             MatrixS M22_2 = B.multiplyDivRecursive(y12, d11_2, ring);
44.             Element ds = d12.multiply(d21, ring).divide(d11, ring);
45.             MatrixS M22_3 = M22_2.multiplyLeftI
46.                 (Array.involution(m21.Ei, (int) finalN.value));
47.             outData[0] = M22_2;
48.             outData[1] = ds;
49.             outData[2] = M22_3;
50.             break;
51.         }
52.         case(7716): {
53.             MatrixS Q1 = (MatrixS) inData[0];
54.             MatrixS M12_1 = (MatrixS) inData[1];
55.             AdjMatrixS m11 = (AdjMatrixS) inData[2];
56.             Element d21 = inData[3];
57.             Element d11 = inData[4];
58.             MatrixS y12 = (MatrixS) inData[5];
59.             AdjMatrixS m12 = (AdjMatrixS) inData[6];
60.             outData[0] = (
61.             ((Q1.subtract((M12_1.multiplyLeftI(m11.Ei).multiplyByNumber(d21, ring)),
    ring))
62.                 .divideByNumber(d11, ring).multiplyRecursive(y12, ring))
63.                     .add((m12.S).multiplyByNumber(d21, ring), ring)
64.             )
65.                     .divideByNumber(d11, ring);
66.             break;
67.         }
68.         // кейси 7714, 7717-7719, 7722-7724
69.         // було пропущено для компактності прикладу
70.         case(7725): {
71.             MatrixS L = (MatrixS) inData[0];
72.             MatrixS F = (MatrixS) inData[1];
73.             MatrixS G = (MatrixS) inData[2];
74.             Element d12 = inData[3];
75.                             outData[0]    =    (L.add(F.multiplyRecursive(G,    ring),
    ring)).divideByNumber(d12, ring);
76.             break;
77.         }
78.     }
79. }
```

На цьому прикладі можна побачити використаний шанс згрупувати ті, що мають на вхід однакового типу параметри та вираховують ту ж саму операцію над цими даними.

Вхідна функція також залежить від варіації дропа, що буде обраховуватися, але для всіх кейсів є спільним те, що вони повинні ініціювати 2 матриці, частини яких будуть множитися в рекурсивних дропах, зазначених в аміні:

```
1.  @Override
2.  public MatrixS[] inputFunction(Element[] input, Amin amin, Ring ring) {
3.      MatrixS[] res = new MatrixS[8];
4.      MatrixS v1;
```

```
5.      MatrixS v2;
6.      switch (key) {
7.          case(7702):
8.          case(7712):
9.          case(7713):
10.         case(7718):
11.         case(7724):
12.         default: {
13.             v1 = (MatrixS) input[0];
14.             v2 = (MatrixS) input[1];
15.             break;
16.         }
17.         case(7708):
18.         case(7720): {
19.             v1 = ((AdjMatrixS) input[0]).A;
20.             v2 = ((AdjMatrixS) input[1]).A;
21.             break;
22.         }
23.         // кейси 7703, 7705, 7707, 7709-7711
24.         // 7714, 7717, 7719, 7721-7723, 7725
25.         // було пропущено для компактності прикладу
26.         case(7716): {
27.             MatrixS Q1 = (MatrixS) inData[0];
28.             MatrixS M12_1 = (MatrixS) inData[1];
29.             AdjMatrixS m11 = (AdjMatrixS) inData[2];
30.             Element d21 = inData[3];
31.             Element d11 = inData[4];
32.                v1 = (Q1.subtract((M12_1.multiplyLeftI(m11.Ei).multiplyByNumber(d21,
    ring)), ring))
33.                     .divideByNumber(d11, ring);
34.             v2 = (MatrixS) inData[5];
35.             break;
36.         }
37.     }
38.     Array.concatTwoArrays(v1.split(), v2.split(), res);
39.     return res;
40. }
```

Оскільки в деяких варіаціях певні розрахунки можуть відбуватися паралельно до рекурсивного множення, то доречно перевантажити метод *independentCalc()*:

```
1.  @Override
2.  public void independentCalc(Ring ring, Amin amin){
3.      switch (key){
4.          case(0): case(1):
5.          case(2): case(7702):
6.          case(7703): case(7707):
7.          case(7708): case(7709):
8.          case(7710): case(7711):
9.          case(7713): case(7714):
10.         case(7717): case(7718):
11.         case(7719): case(7720):
12.         case(7721): case(7722):
13.         case(7723): case(7724):
14.         case(7725): break;
15.         case(7705): {
16.             MatrixS s1 = ((MatrixS) inData[0]).multiplyByNumber(inData[1], ring);
17.             amin.resultForOutFunction[4] = s1;
18.             break;
19.         }
20.         case(7712): {
```

```
            Element ds = inData[3].multiply(inData[4], ring).divide(inData[2], ring);
            amin.resultForOutFunction[4] = ds;
            break;
        }
        case(7716): {
            MatrixS S12 = ((AdjMatrixS) inData[6]).S;
            Element d21 = inData[3];
            amin.resultForOutFunction[4] = S12.multiplyByNumber(d21, ring);
            break;
        }
    }
    return;
}
```

Перевірка *isItLeaf()* також залежить від варіації дропа, оскільки не для кожної варіації першим вхідним аргументом дропу буде об'єкт типу *MatrixS:*

```
@Override
public boolean isItLeaf() {
    MatrixS ms;
    switch(key){
        case(0): case(1):
        case(2): case(7702):
        case(7705): case(7711):
        case(7712): case(7713):
        case(7714): case(7716):
        case(7717): case(7718):
        case(7719): case(7721):
        case(7722): case(7724):
        case(7725):
        default: {
            ms = (MatrixS) inData[0];
            break;
        }
        case(7703):
        case(7707):
        case(7709): {
            ms = (MatrixS) inData[1];
            break;
        }
        case(7708):
        case(7720):{
            ms = ((AdjMatrixS) inData[0]).A;
            break;
        }
        case(7710):
        case(7723): {
            ms = ((AdjMatrixS) inData[0]).S;
            break;
        }
    }
    return (ms.size <= leafSize);
}
```

Вихідна функція дропу залежить від варіації. Спільним для всіх варіацій є те, що з рекурсивних обрахуванків повернуто 4 матриці, яких на початку методу об'єднують, тим самим отримуючи результат множення двох

матриць. Також спільним є те, що кожна варіація повертає масив результатів, хоча для кожної варіації різна довжина цього масиву:

```
1.  @Override
2.  public Element[] outputFunction(Element[] input, Ring ring) {
3.      MatrixS[] resmat = new MatrixS[4];
4.      for (int i = 0; i < 4; i++) {
5.          resmat[i] = (MatrixS) input[i];
6.      }
7.      Element[] res = new Element[outputDataLength];
8.      switch (key){
9.          case(7708):
10.         case(7713):
11.         case(7720): {
12.             res = new MatrixS[]{MatrixS.join(resmat)};
13.             break;
14.         }
15.         case(7702):
16.         case(7707):
17.         case(7718): {
18.             Element d = inData[2];
19.             MatrixS result1 = MatrixS.join(resmat)
20.                 .divideByNumber(d.negate(ring), ring);
21.             res = new MatrixS[]{result1};
22.             break;
23.         }
24.         case(7712): {
25.             Element d11 = inData[2];
26.             AdjMatrixS m21 = (AdjMatrixS) inData[5];
27.             Element finalN = inData[6];
28.             MatrixS M22_2 = MatrixS.join(resmat)
29.             .divideByNumber(d11
30.                     .multiply(d11, ring), ring);
31.             MatrixS M22_3 = M22_2.multiplyLeftI
32.                 (Array.involution(m21.Ei, (int) finalN.value));
33.             res = new Element[]{M22_2, input[4], M22_3};
34.             break;
35.         }
36.         // кейси 7703, 7705, 7709-7711, 7716-7717,
37.         // 7719, 7721-7722, 7724-7725
38.         // було пропущено для компактності прикладу
39.         case(7714):
40.         case(7723): {
41.             Element d0 = inData[2];
42.             Element d12 = inData[3];
43.             MatrixS K2 = (MatrixS) inData[4];
44.             Element d11 = inData[5];
45.             MatrixS s1 = MatrixS.join(resmat)
46.                 .divideMultiply(d0, d12, ring);
47.             MatrixS s2 = s1.add(K2, ring);
48.             MatrixS result1 = s2.divideByNumber
49.                 (d11.negate(ring), ring);
50.             res = new MatrixS[]{result1};
51.             break;
52.         }
53.     }
54.     return res;
55. }
```

4.3. Клас MatrSAdjMatrix

Реалізація перевірки *isItLeaf()* є ідентичною до реалізації у класі *MatrSMultiplyScalar*, тому її тут опущено.

Так само як і у *MatrSMultiplyScalar*, усі потрібні поля ініціалізуються в конструкторі класу:

```
1.  public MatrSAdjMatrix() {
2.      arcs = _arcs;
3.      type = 7701;
4.      number = cnum++;
5.      inputDataLength = 2;
6.      outputDataLength = 3;
7.      inData = new Element[inputDataLength];
8.      outData = new Element[outputDataLength];
9.      resultForOutFunctionLength = 14;
10. }
```

Одною з найскладніших частин у реалізації цього класу було прописання залежностей між дропами:

```
1.  private static int[][] _arcs = new int[][]{
2.      {1,0,0,  2,2,0,  3,1,1,  5,3,0,  5,2,2,
3.      14,2,0,  1,4,1,  2,4,2,  3,4,2,  3,5,3,
4.      5,4,5,  12,5,6,  14,4,2},
5.      // 0 inputFunction; I(M, d0), O( M[0], M[1], M[2], M[3], d0, finalN)
6.      {2,1,1,  3,0,0,  4,2,1,  5,2,1,  5,0,4,
7.      6,2,1,  7,0,0,  7,2,2,  8,0,1,  10,0,0,
8.      10,2,2,  11,2,3,  12,2,2,  14,0,1,  14,2,5,
9.      16,0,2,  16,2,4,  17,0,2,  17,2,4,  23,0,0,
10.     23,2,2,  26,0,10},
11.     // 1, here is done y11; I( M[0], d0  ), O ( m11, y11, d11 )
12.     {4,0,0  }, // 2; I( M[2], y11, d0 ), O ( M21_1 )
13.     {5,0,3,  6,1,0,  16,0,1,  17,0,1}, // 3; I( m11, M[1], d0, finalN ), O ( M12_1,
    M12_2 )
14.     {7,1,1,  9,0,0,  10,0,1,  12,2,3,  12,0,5,
15.     16,2,3,  19,0,1,  20,0,1,  22,0,2,  22,2,5,
16.     23,0,1,  23,2,5,  26,2,6,  26,0,8},
17.     // 4; I( M21_1, d11 ), O ( m21, y21, d21 )
18.     {9,0,1,  11,0,0,  13,0,1}, // 5; I( M[3], d11, M[2], M12_1, m11, d0 ), O ( M22_1
    )
19.     {8,0,0,  11,0,2,  12,1,1,  12,2,4,  14,2,3,
20.     16,1,5,  16,0,6,  17,0,3,  24,2,2,  25,2,3,
21.     26,0,11}, // 6; I( M12_2, d11 ), O ( m12, y12, d12 )
22.     {26,0,4}, // 7; I( m11, y21, d11 ), O ( M11_2 )
23.     {11,0,1,  17,0,0}, // 8; I( m12, m11 ), O ( A1 )
24.     {12,0,0}, // 9; I( m21, M22_1 ), O ( B )
25.     {13,0,0}, // 10; I( m11, m21, d11 ), O ( Q )
26.     {14,0,4}, // 11; I( M22_1, A1, m12, d11 ), O ( K2 )
27.     {15,2,0,  15,1,1,  18,1,2,  19,0,0,  19,1,3,
28.     21,1,3,  22,0,1,  22,1,4},
29.     // 12; I( B, y12, d11, d21, d12, m21, finalN ), O ( M22_2, ds, M22_3 )
30.     {16,0,0}, // 13; I( Q, M22_1 ), O ( Q1 )
31.     {24,0,1,  25,0,2}, // 14; I( M[2], m11, d0, d12, K2, d11 ), O ( G )
32.     {17,2,5,  18,1,1,  19,1,2,  19,0,4,  20,0,0,
33.      21,0,2,  22,0,3,  23,2,3,  26,2,5,  26,0,12}, // 15; I( M22_3, ds ), O (
    m22,y22,d22 )
```

```
    {18,0,0,  21,0,0}, // 16; I( Q1, M12_1, m11, d21, d11, y12, m12 ), O ( M12_2_new )
    {25,0,0}, // 17; I( A1, M12_1, m11, m12, d11, d22 ), O ( L )
    {26,0,7}, // 18; I( M12_2_new , y22, ds ), O ( M12_3 )
    {26,0,9}, // 19; I( M22_2, m21, y22, ds, m22 ), O ( M22_3_new )
    {21,0,1,  22,0,0}, // 20; I( m22, m21 ), O ( A2 )
    {23,0,4}, // 21; I( M12_2_new, A2, m22, ds ), O ( K1 )
    {24,0,0,  26,0,3}, // 22; I( A2, M22_2, m21, m22, ds, d21 ), O ( P )
    {25,0,1,  26,0,1}, // 23; I( m11, m21, d11, d22, K1, d21 ), O ( F )
    {26,0,2}, // 24; I( P, G, d12 ), O ( P1 )
    {26,0,0}, // 25; I( L, F, G, d12 ), O ( F1 )
    {}, // 26 outputFunction;
    //I( F1, F, P1, P, M11_2, d22, d21, M12_3, m21, M22_3_new, m11, m12, m22, finalN ),
    // O ( m, y, d)
};
```

Де *I()* - вхідні параметри дропу, а *O()* – вихідні.

Наступним чином декларовано метод *doAmin()*:

```
@Override
public ArrayList<Drop> doAmin() {
    ArrayList<Drop> amin = new ArrayList<Drop>();
    amin.add(new MatrSAdjMatrix());
    amin.add(new MatrSMult4());
    amin.get(1).key = 7702;
    amin.add(new MatrSMult4());
    amin.get(2).key = 7703;
    amin.add(new MatrSAdjMatrix());
    amin.add(new MatrSMult4());
    amin.get(4).key = 7705;
    amin.add(new MatrSAdjMatrix());
    amin.add(new MatrSMult4());
    amin.get(6).key = 7707;
    amin.add(new MatrSMult4());
    amin.get(7).key = 7708;
    amin.add(new MatrSMult4());
    amin.get(8).key = 7709;
    amin.add(new MatrSMult4());
    amin.get(9).key = 7710;
    amin.add(new MatrSMult4());
    amin.get(10).key = 7711;
    amin.add(new MatrSMult4());
    amin.get(11).key = 7712;
    amin.add(new MatrSMult4());
    amin.get(12).key = 7713;
    amin.add(new MatrSMult4());
    amin.get(13).key = 7714;
    amin.add(new MatrSAdjMatrix());
    amin.add(new MatrSMult4());
    amin.get(15).key = 7716;
    amin.add(new MatrSMult4());
    amin.get(16).key = 7717;
    amin.add(new MatrSMult4());
    amin.get(17).key = 7718;
    amin.add(new MatrSMult4());
    amin.get(18).key = 7719;
    amin.add(new MatrSMult4());
    amin.get(19).key = 7720;
    amin.add(new MatrSMult4());
    amin.get(20).key = 7721;
    amin.add(new MatrSMult4());
```

```
43.     amin.get(21).key = 7722;
44.     amin.add(new MatrSMult4());
45.     amin.get(22).key = 7723;
46.     amin.add(new MatrSMult4());
47.     amin.get(23).key = 7724;
48.     amin.add(new MatrSMult4());
49.     amin.get(24).key = 7725;
50.     return amin;
51. }
```

Можна побачити, що 0, 3, 5 й 14 індекси є рекурсивними дропами даного класу, а інші – варіації множення. Після додавання кожної варіації множення до аміну вказано яку саме варіацію вона обчислює шляхом вказування значення *key* для цього дропу.

Наступним чином реалізовано методи *sequentialCalc()*:

```
1.  @Override
2.  public void sequentialCalc(Ring ring) {
3.      MatrixS m = (MatrixS) inData[0];
4.      Element d0 = inData[1];
5.      AdjMatrixS adjM = new AdjMatrixS(m, d0,  ring);
6.      Element resD = adjM.Det;
7.      MatrixS y = adjM.S.ES_min_dI(resD, adjM.Ei, adjM.Ej, ring);
8.      outData[0] = adjM;
9.      outData[1] = y;
10.     outData[2] = resD;
11. }
```

Як видно, на відмінність від попередніх двох класів, цей дроп повертає як результат масив з 3 елементів.

У вхідній функції, окрім 4 частин вхідної матриці, в рекурсивні дропи також передаються визначник та розмірність частин вхідної матриці. Останнє також зразу передається у вхідні параметри вихідної функції:

```
1.  @Override
2.  public Element[] inputFunction(Element[] input, Amin amin, Ring ring) {
3.      MatrixS m = (MatrixS) input[0];
4.      MatrixS[] splitted = m.split();
5.      Element[] res = new Element[6];
6.      System.arraycopy(splitted, 0, res, 0, 4);
7.      res[4] = input[1];
8.      // todo ???
9.      int N = m.size;
10.     int finalN = N >>> 1;
11.     res[5] = new Element(finalN);
12.     amin.resultForOutFunction[13] = new Element(finalN);
13.     return res;
14. }
```

У вихідній функції за допомогою тих аргументів, що були обраховані в рекурсивних дропах, буде обчислено приєднану матрицю, ешелону форму

матриці та визначник. Для обчислень в рекурсивних дропах MatrSAdjMatrix також обчислюється матриця Y:

```
1.  @Override
2.  public Element[] outputFunction(Element[] input, Ring ring) {
3.      MatrixS[] aParts = new MatrixS[4];
4.      for (int i = 0; i < 4; i++) {
5.          aParts[i] = (MatrixS) input[i];
6.      }
7.      MatrixS A = MatrixS.join(aParts);
8.      MatrixS[] sParts = new MatrixS[4];
9.      MatrixS M11_2 = (MatrixS) input[4];
10.     Element d22 = input[5];
11.     Element d21 = input[6];
12.     MatrixS M12_3 = (MatrixS) input[7];
13.     AdjMatrixS m21 = (AdjMatrixS) input[8];
14.     MatrixS M22_3_new = (MatrixS) input[9];
15.     AdjMatrixS m11 = (AdjMatrixS) input[10];
16.     AdjMatrixS m12 = (AdjMatrixS) input[11];
17.     AdjMatrixS m22 = (AdjMatrixS) input[12];
18.     Element finalN = input[13];
19.     int N = (int) finalN.value;
20.     sParts[0] = M11_2.multiplyDivide(d22, d21, ring);
21.     sParts[1] = M12_3;
22.     sParts[2] = m21.S.multiplyDivide(d22, d21, ring);
23.     sParts[3] = M22_3_new;
24.     MatrixS S = MatrixS.join(sParts);
25.     int[] Ei = new int[m11.Ei.length+m12.Ei.length+m21.Ei.length+m22.Ei.length];
26.     int[] Ej = new int[Ei.length];
27.     int j=0;
28.     for (int i = 0; i < m11.Ei.length;) {Ei[j] = m11.Ei[i];
29.         Ej[j++] = m11.Ej[i++];  }
30.     for (int i = 0; i < m12.Ei.length;) {Ei[j] = m12.Ei[i];
31.         Ej[j++] = m12.Ej[i++]+N;}
32.     for (int i = 0; i < m21.Ei.length;) {Ei[j] = m21.Ei[i]  +N;
33.         Ej[j++] = m21.Ej[i++];  }
34.     for (int i = 0; i < m22.Ei.length;) {Ei[j] = m22.Ei[i]  +N;
35.         Ej[j++] = m22.Ej[i++]+N;}
36.     // результат
37.     AdjMatrixS res = new AdjMatrixS(A, Ei, Ej, S, d22);
38.     Element d = res.Det;
39.     MatrixS y = res.S.ES_min_dI(d, res.Ei, res.Ej, ring);
40.     return new Element[] {res, y, d};
41. }
```

Розділ 5. Проведення експериментів

Експерименти були проведені на матрицях зі сторонами 128, 256, 512. Були використані матриці з наступною конфігурацією: розмір (можливі використані значення вказані вище), leaf size (найоптимальніший буде вказуватися в таблицях нижче), щільність 100%, варіація чисел усередині матриць зі значення не більше ніж 2^5. Додатково було проведено експеримент на матриці зі стороною 512 та різною щільністю заповнення матриці: 3%, 30%, 100%. Експерименти були проведені на персональному комп'ютері із 4 ядрами.

Далі наведено приклад команди запуску:

```
1. mpirun --hostfile /Users/mldtsv/univ/threads/hostfile -np 4 java -cp
   /Users/mldtsv/openmpi/lib/mpi.jar:target/qr-test.jar \
2. -Xmx4g com/mathpar/parallel/dap/adjmatrix/MatrixS/Tests/MatrSAdjMatrixTest -size=512
   -leaf=256 -nocheck
```

Флаг -nocheck повідомляє, що не треба тестувати чи правильно була обрахована матриця. Доречно сказати, що тестування виконується завдяки методу класу DAPTest.checkResult. Алгоритм цієї перевірки полягає в наступному: після отримання результатів вирахованих паралельним шляхом, вираховується послідовним шляхом (імплементація - клас який вже існував у пакунку DAP й був протестований) матриця й результати порівнюються, а саме порівнюються приєднана матриця, ешелона форма та визначник. Альтернативою для перевірки правильності обчислень є варіант, коли обчислена паралельним шляхом приєднана матриця ділиться на визначник й порівнюється з вже відомою оберненою матрицею. Оскільки процес тестування значно сповільнює процес виконання, під час проведення серії експериментів було використано флаг -nocheck. Було проведено перевірку на матрицях розміру 128 й доведено таким чином правильність обрахунку.

Далі наведено ще один приклад команди запуску, але зі вказанням щільності матриці, що повинна бути сгенерована:

```
1. mpirun --hostfile /Users/mldtsv/univ/threads/hostfile -np 4 java -cp
   /Users/mldtsv/openmpi/lib/mpi.jar:target/qr-test.jar \
2. -Xmx4g com/mathpar/parallel/dap/adjmatrix/MatrixS/Tests/MatrSAdjMatrixTest -size=512
   -leaf=256  -density=30 -nocheck
```

Результати виконання програми на матриці розміром 128:

N= 128							
proc	1	2	3	4	5		best leaf size
1	981	982	972	1162	1129	1045.2	64
2	1186	1192	1249	1287	1206	1224	128
3	1164	1227	1280	1355	1147	1234.6	128
4	1266	1275	1215	1194	1399	1269.8	128

Рисунок 5 Результати виконання програми на матриці розміром 128

N=128	Best leaf size 128			
processors amount	1	2	3	4
time average (ms)	1045.2	1224	1234.6	1269.8

У середньому час виконання на 4 процесорах у порівнянні з 1 на -21,489% швидший.

Результати виконання програми на матриці розміром 256:

N= 256							
proc	1	2	3	4	5		best leaf size
1	21493	21411	21351	21424	21466	21429	256
2	21506	20310	21496	20876	21670	21171.6	128
3	18107	14954	17381	15228	14415	16017	64
4	26350	31368	25848	26178	27517	27452.2	128

Рисунок 6 Результати виконання програми на матриці розміром 256

N=256	Best leaf size 128			

processors amount	1	2	3	4
time average (ms)	21429	21171.6	16017	27452.2

У середньому час виконання на 4 процесорах у порівнянні з 1 на -28,108% швидший.

Результати виконання програми на матриці розміром 512:

N= 512							
proc	1	2	3	4	5		best leaf size
1	957684	960123	954499	1023110	1004320	979947.2	256
2	650562	668590	730615	672888	699108	684352.6	256
3	610736	736832	673219	653341	645512	663928	256
4	597304	601084	727947	620566	667456	642871.4	256

Рисунок 7 Результати виконання програми на матриці розміром 512

N=512	Best leaf size 256			
processors amount	1	2	3	4
time average (ms)	979947.2	684352.6	663928	642871.4

У середньому час виконання на 4 процесорах у порівнянні з 1 на 34,397% швидший.

Результати виконання програми на матриці розміром 512, з використанням 4 ядер та різною щільністю матриць:

N=512	processsors amount = 4		
Density (%)	3	30	100
time (ms)	390767	601061.2	617084

З експериментів видно, що архітектура системи побудована на створенні обчислювального графа та рекурсивного обчислення дропів. Чим менший leaf size у порівнянні із розміром сторони матриці, тим більшу кількість дропів створить система. Чим більшу кількість дропів створить система, тим більша кількість ядер потрібна комп'ютеру, щоб вираховувати їх без очікування цих дропів у черзі. Саме тому при розрахунках на 4-ядерному комп'ютері найкращими leaf size є або сам розмір матриці (для матриці розміром 128) або у два рази менше число. Також можна зазначити, що такий метод обрахунку не оптимальний для найменших матриць й інші спрощені методи обрахують завдання швидше. Можна побачити по результатам, що з матрицями маленької щільності (такі екземпляри зустрічаються найчастіше на практиці) система працює та показує кращі результати, ніж виконання обрахунку з матрицями більшої щільності.

Висновки

Було розглянуто алгоритм обчислення приєднаної матриці для знаходження оберненої матриці. Запроваджено та проаналізовано блочний алгоритм обчислення приєднаної матриці. Імплементовано реалізації за допомогою засобів стандартної бібліотеки Java та проведені над цими імплементаціями експерименти. Розглянуто архітектуру системи динамічного управління завданнями DAP. Описані елементи цієї системи: дроп, амін, транспорт, пайн, вокзал, аеродром, термінал. Порівняно різновиди потоків, що використовуються у програмі: потік-диспетчер та обчислювальний потік. Описано роль та функціональність кожного виду. Імплементовано дропи для обчислення приєднаної матриці. Прокоментована імплементація дропів та розглянуті фрагменти коду. Проведено експерименти та встановлено ефективність використання даного рішення при обчисленні розріджених матриць. Встановлено взаємозв'язок між параметром leaf size та кількістю дропів, що створюється для обчислення. З'ясовано, що при збільшенні кількості ядер на комп'ютері, на якому здійснюється запуск системи, час виконання програми зменшиться у порівнянні із запуском на комп'ютері з меншою кількістю ядер. У майбутніх дослідженнях розглядатимуться експерименти на комп'ютерах зі значно більшою кількістю ядер та шляхи для оптимізації та пришвидшення системи.

Список використаних джерел

1. Борнс А. Суперкомп'ютери у 1990 роках / А. Борнс, Ч. Бендер // Суперкомп'ютери в інженерному аналізі / А. Борнс, Ч. Бендер. – Нью-Йорк: Марсел Деккер, 1992. – С. 1–18.
2. Адаптивні лінійні вирішувачі та власні розв'язувачі, Джек Донгарра [Електронний ресурс] //Аргоннська національна лабораторія. – 2019. – Режим доступу до ресурсу: https://extremecomputingtraining.anl.gov/files/2019/08/ATPESC_2019_Track-5_1_8-5_830am_Dongarra-Adaptive_Linear_Solvers_and_Eigensolvers.pdf.

3. *Евгений Ильченко*. Об одном алгоритме управления параллельными вычислениями с децентрализованным вычислением. [Електронний ресурс]. – режим доступу: https://cyberleninka.ru/article/n/ob-odnom-algoritme-upravleniya-parallelnymi-vychisleniyami-s-detsentralizovannym-upravleniem
4. *Евгений Ильченко*. Об эффективном методе распараллеливания блочных рекурсивных алгоритмов. [Електронний ресурс]. – режим доступу: https://cyberleninka.ru/article/n/ob-effektivnom-metode-rasparallelivaniya-blochnyh-rekursivnyh-algoritmov
5. *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
6. *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
7. *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.
8. *Malaschonok G. I., Ilchenko E.* Recursive Matrix Algorithms in Commutative Domain for Cluster with Distributed Memory. 2018 Ivannikov Memorial Workshop (IVMEM), (Yerevan, Armenia, 3-4May 2018). P. 40–47. [Електронний ресурс]. – режим доступу: arxiv.org/abs/1903.04394. Publisher:IEEE: https://ieeexplore.ieee.org/document/86363423.
9. A High Performance Message Passing Library[Електронний ресурс]. – Режим доступу до ресурсу: https://www.open-mpi.org.