МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №8

з курсу «Технології паралельних обчислень»

на тему: «Розробка алгоритмів для розподілених систем клієнт-серверної архітектури»

Викладач:
Дифучина О.Ю.

Виконала:
студентка 3 курсу
групи ІП-14 ФІОТ
Радзівіло Валерія

Київ-2023

Завдання:

Розробити веб-застосування клієнт-серверної архітектури, що реалізує алгоритм множення матриць (або інший обчислювальний алгоритм, який був Вами реалізований іншими методами розподілених обчислень в рамках курсу «Паралельні та розподілені обчислення») на стороні сервера з використанням паралельних обчислень. Розгляньте два варіанти реалізації 1) дані для обчислень знаходяться на сервері та 2) дані для обчислень знаходяться на клієнтській частині застосування. 60 балів.

Дослідити швидкість виконання запиту користувача при різних обсягах даних. 20 балів.

Порівняти реалізацію алгоритму в клієнт-серверній системи та в розподіленій системі з рівноправними процесорами. 20 балів.

Виконання завдання:

Завдання 1

Цей проект використовує декілька технологій для розробки веб-додатку, який виконує операції з матрицями. Ось основні технології, які використовуються: Java: Основна мова програмування для серверної частини додатку. Використовується для написання бізнес-логіки, включаючи алгоритми обчислення матриць. Spring Boot: Фреймворк для створення веб-додатків на Java. Використовується для створення RESTful API, які взаємодіють з клієнтською частиною додатку. Maven: Інструмент для керування проектами на Java, який використовується для збірки проекту, керування залежностями та іншого. JavaScript: Мова програмування, яка використовується для клієнтської частини додатку. React: JavaScript-фреймворк для створення інтерфейсів користувача. Використовується для створення клієнтської частини додатку. NPM: Менеджер пакетів для JavaScript, який використовується для керування залежностями клієнтської частини додатку.

В цьому проекті використовується алгоритм Fox для множення матриць, який реалізований на Java. Веб-сервіс, створений за допомогою Spring Boot, надає API для взаємодії з цим алгоритмом. Клієнтська частина додатку, створена на React, використовує ці API для виконання операцій з матрицями.

Реалізація №1 - дані для обчислень знаходяться на сервері

```
@GetMapping("/{size}")
public List<int[][]> showMatrices(@PathVariable String size) {
    int s = Integer.parseInt(size);
    Matrix matrix1 = new Matrix(s, s);
    Matrix matrix2 = new Matrix(s, s);
    matrix1.generateRandomMatrix();
    matrix2.generateRandomMatrix();
    return Arrays.asList(matrix1.matrix, matrix2.matrix);
}
```

Метод «showMatrices», який використовується в контролері Spring Boot для обробки HTTP GET запитів на шлях «/{size}». «@PathVariable String size» означає, що значення «size» береться з частини URL, яка відповідає «{size}».

В методі «showMatrices» виконуються наступні дії: Значення «size», отримане з URL, перетворюється в ціле число «s» за допомогою «Integer.parseInt(size)». Створюються дві матриці «matrix1» та «matrix2» розміром «s x s» за допомогою конструктора класу «Matrix». Для обох матриць викликається метод «generateRandomMatrix()», який заповнює матриці випадковими числами. Метод повертає список цих двох матриць за допомогою «Arrays.asList(matrix1.matrix, matrix2.matrix)».

Отже, цей метод використовується для генерації двох випадкових матриць заданого розміру та відправки їх у відповідь на HTTP GET запит.

Файл `index.html` використовує React для створення інтерактивного веб-інтерфейсу. Ось основні елементи UI: Кнопка "Get Matrices from Java": Коли користувач натискає цю кнопку, виконується запит до сервера за адресою `http://127.0.0.1:8080/` з поточним розміром матриці. Отримані матриці відображаються на сторінці.

```
<button onClick={() => {
    fetch('http://127.0.0.1:8080/' + matrixSize.valueOf())
        .then(response => response.json())
        .then(data => {
            setMatrices(data);
            setDataSourceInput(false);
        })
}}>Get Matrices from Java
</button>
```

Поля вводу "Size" та "Thread count": Користувач може ввести розмір матриці та кількість потоків для обчислень.

```
  <label>
        Size:
        <input type="number" name="size" min="1" max="500"
defaultValue={matrixSize}
```

```
                onChange={(e) => {
                    console.log(e.target.value);
                    setMatrixSize(e.target.value);
                }}/>
    </label>
    <br/> <label>
    Thread count:
    <input type="number" name="size" min="1" max="10"
defaultValue={threadCount}
           onChange={(e) => {
               console.log("New thread count : " + e.target.value);
               setThreadCount(e.target.value);
           }}/>
</label>
    <br/>
```

Відображення матриць: Якщо користувач не вводить матриці вручну, відображаються матриці, отримані з сервера. Кожна матриця відображається в таблиці.

```
{matrices.length > 0 && !dataSourceInput &&
    matrices.map((matrix, matrixIndex) => (
        <div key={matrixIndex}>
            <h2>Matrix {matrixIndex + 1}</h2>
            <table>
                <tbody>
                {matrix.map((row, rowIndex) => (
                    <tr key={rowIndex}>
                        {row.map((item, itemIndex) => (
                            <td key={itemIndex}>{item}</td>
                        ))}
                    </tr>
                ))}
                </tbody>
            </table>
        </div>
    ))
}
```

Кнопка "Multiply Matrices": Коли користувач натискає цю кнопку, виконується множення матриць. Результат відображається на сторінці.

```
{!dataSourceInput && <button onClick={
    () => {
        console.log("Multiply Matrices")

        const payload = new MatrixPayload(matrices[0], matrices[1],
threadCount.valueOf());
        fetch('http://127.0.0.1:8080/multiply', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(payload),

        })
            .then(response => {
                if (!response.ok) {
                    throw new Error("HTTP error " + response.status);
                }
                return response.json();
```

```
            })
            .then(data => {
                // output matrix result data on the screen
                setResultMatrix(data);

                console.log(data);
            });

    }
}>Multiply Matrices
</button>}
```

Відображення результату: Результат множення матриць відображається в таблиці під кнопкою "Multiply Matrices".

```
{resultMatrix.length > 0 && (
    <div>
        <h2>Result Matrix</h2>
        <table>
            <tbody>
            {resultMatrix.map((row, rowIndex) => (
                <tr key={rowIndex}>
                    {row.map((item, itemIndex) => (
                        <td key={itemIndex}>{item}</td>
                    ))}
                </tr>
            ))}
            </tbody>
        </table>
    </div>
)}
```

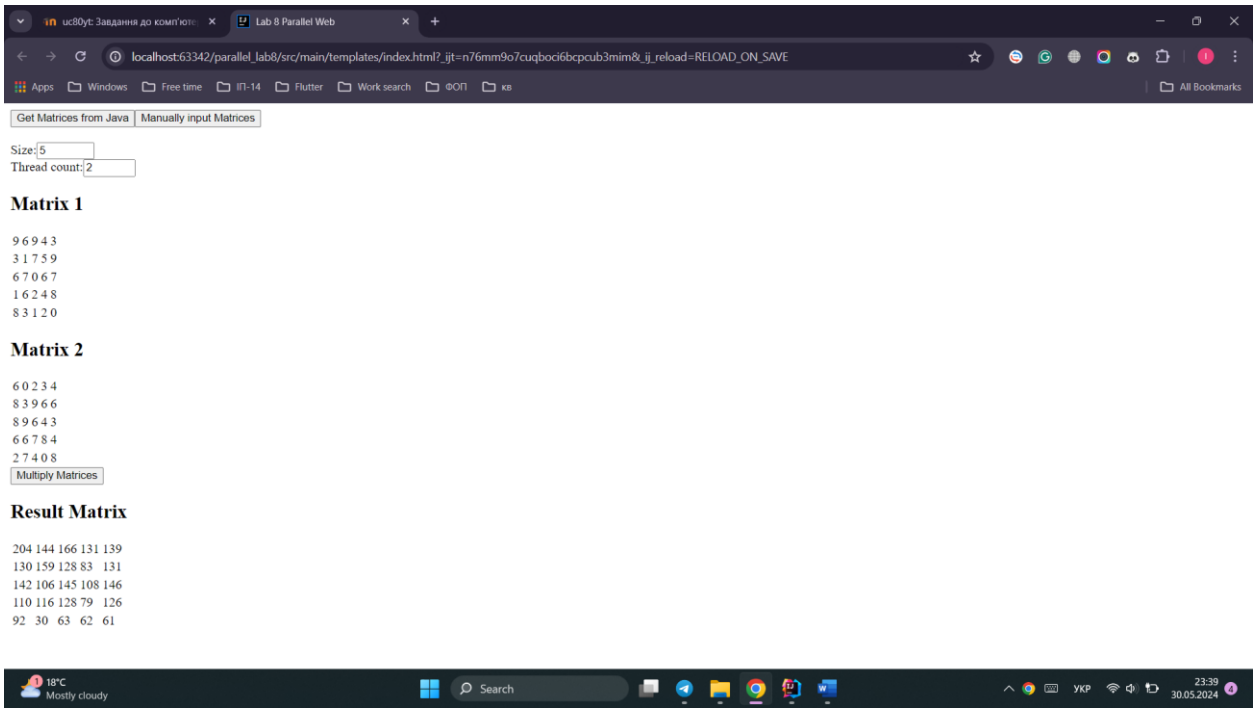
Цей інтерфейс дозволяє користувачу взаємодіяти з сервером, отримувати матриці, встановлювати розмір матриці та кількість потоків, а також отримувати результат множення матриць.

Алгоритм Фокса взятий з другої лабораторної роботи без вагомих змін. Ось так проходить множення матриць зі сторони Spring.

```
@PostMapping("/multiply")
public int[][] multiplyMatrices(@RequestBody Map data) {
    final MatrixPayload payload = MatrixPayload.fromMap(data);
    Matrix firstMatrix = payload.getFirstMatrix();
    Matrix secondMatrix = payload.getSecondMatrix();
    FoxAlgorithm algorithm = new FoxAlgorithm(payload.getThreadCount(),
firstMatrix, secondMatrix);
    return algorithm.run(true);
}

@Setter
@Getter
public static class MatrixPayload implements Serializable {
    private Matrix firstMatrix;
    private Matrix secondMatrix;
    private int threadCount;

    MatrixPayload(Matrix firstMatrix, Matrix secondMatrix, int threadCount) {
        this.firstMatrix = firstMatrix;
```

```
        this.secondMatrix = secondMatrix;
        this.threadCount = threadCount;
    }

    public static MatrixPayload fromMap(Map payload) {
        return new MatrixPayload(
                new Matrix((ArrayList) payload.get("firstMatrix")),
                new Matrix((ArrayList) payload.get("secondMatrix")),
                Integer.parseInt(payload.get("threadCount").toString()));

    }

}
```

Виконання:



Реалізація № 2 - дані для обчислень знаходяться на клієнтській частині застосування

Частина коду, що відповідає за введення користувачем даних, включає наступні елементи:

Поля вводу "Size" та "Thread count": Користувач може ввести розмір матриці та кількість потоків для обчислень. Значення, введені користувачем, зберігаються в стані компонента React.

Кнопка "Manually input Matrices": Коли користувач натискає цю кнопку, він може вручну ввести матриці. Це змінює стан `dataSourceInput` на `true`.

```
<button onClick={() => {
    setDataSourceInput(true);
```

```
}}>Manually input Matrices
</button>
```

Форма введення матриць: Якщо `dataSourceInput` дорівнює `true`, відображається форма для введення матриць. Користувач може ввести значення кожного елемента матриці. Після введення матриць користувач може натиснути кнопку "Submit" для відправки форми.

```
{dataSourceInput && <form onSubmit={handleFormSubmit}>
    <h2>Input Matrices</h2>
    {Array.from({length: 2}, (_, i) => (
        <div key={i + "_matrix"}>
            <h2>Matrix {i + 1}</h2>
            {Array.from({length: matrixSize}, (_, rowIndex) => (
                <div key={i + "_matrix_row_" + rowIndex}>
                    {Array.from({length: matrixSize}, (_, columnIndex) => (
                        <label key={columnIndex + "_for_matrix_" + i}>
                            <input type="number" name={`matrix${i + 1}[]`}
required/>
                        </label>
                    ))}
                    <br/>
                </div>
            ))}
        </div>
    ))}
    <input type="submit" value="Submit"/>
</form>}
```

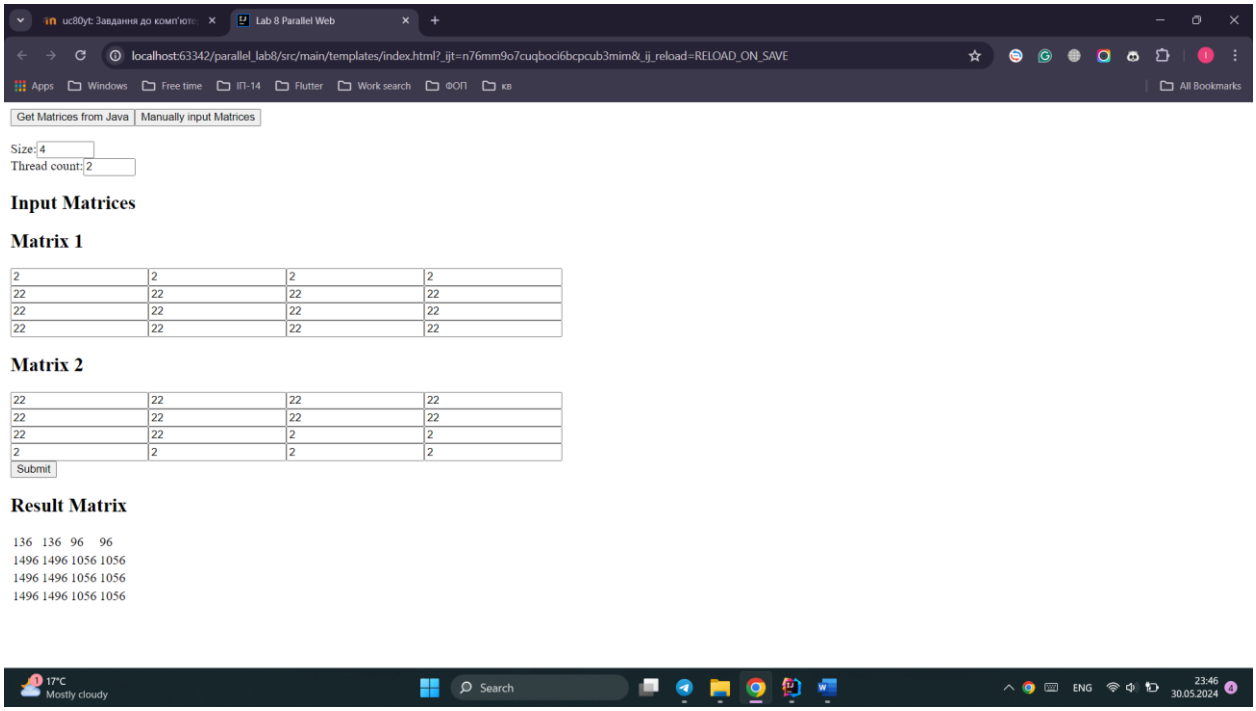
Обробник події `handleFormSubmit`: Коли форма відправляється, викликається ця функція. Вона зчитує дані форми, зберігає введені матриці в стані компонента та виконує запит до сервера для множення матриць.

```
const handleFormSubmit = (event) => {
    event.preventDefault();

    // Get the form data
    const formData = new FormData(event.target);
    const size = matrixSize;
    const matrices = [
        formData.getAll('matrix1[]').map(Number),
        formData.getAll('matrix2[]').map(Number)
    ];

    // Update the state variables
    setMatrixSize(size);
    setInputMatrices(matrices);
    setDataSourceInput(true);

    // Multiply matrices
    const payload = new
MatrixPayload(formData.getAll('matrix1[]').map(Number),
formData.getAll('matrix2[]').map(Number), threadCount.valueOf());
    console.log("Matrix 1")
    console.log(formData.getAll('matrix1[]').map(Number))
    console.log("Matrix 2")
    console.log(formData.getAll('matrix2[]').map(Number))
    fetch('http://127.0.0.1:8080/multiply', {
        method: 'POST',
        headers: {
```

```
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(payload),
    })
        .then(response => response.json())
        .then(data => {
            // output matrix result data on the screen
            setResultMatrix(data);
            console.log(data);
        });
};
```

Отже, користувач може ввести розмір матриці, кількість потоків та самі матриці. Ці дані потім використовуються для виконання обчислень на сервері.

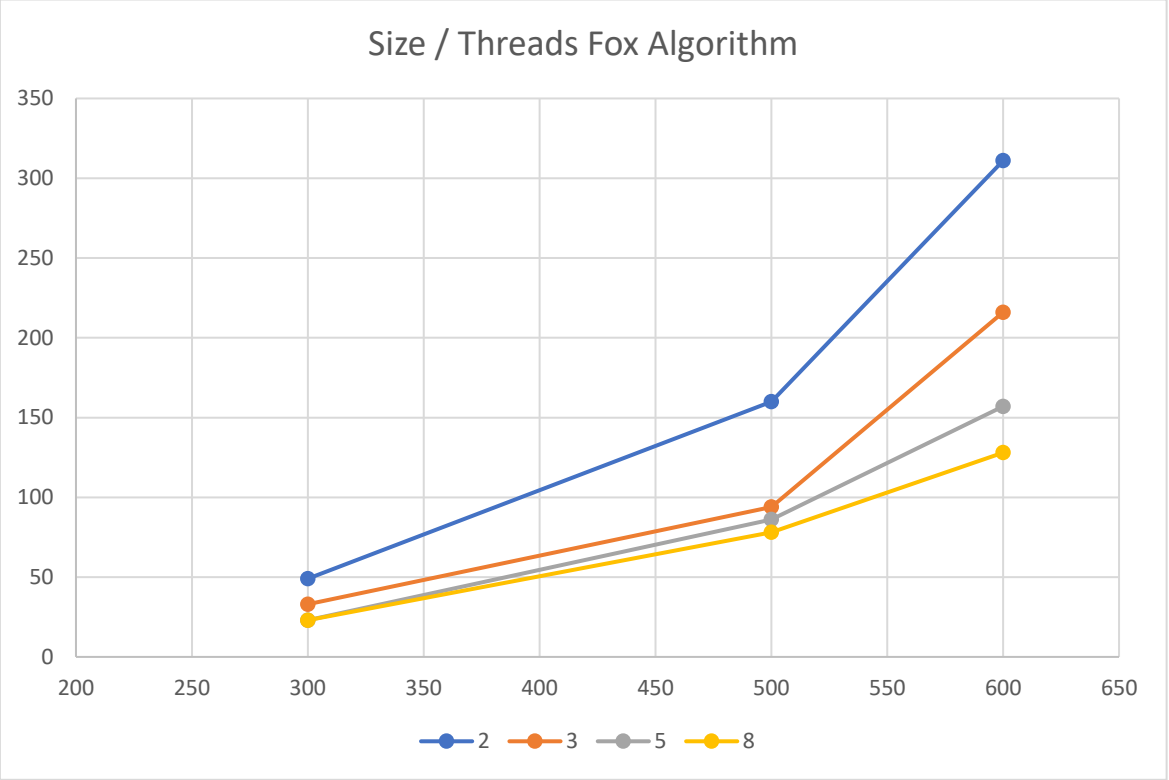
Виконання



Завдання 2 – Дослідити швидкість

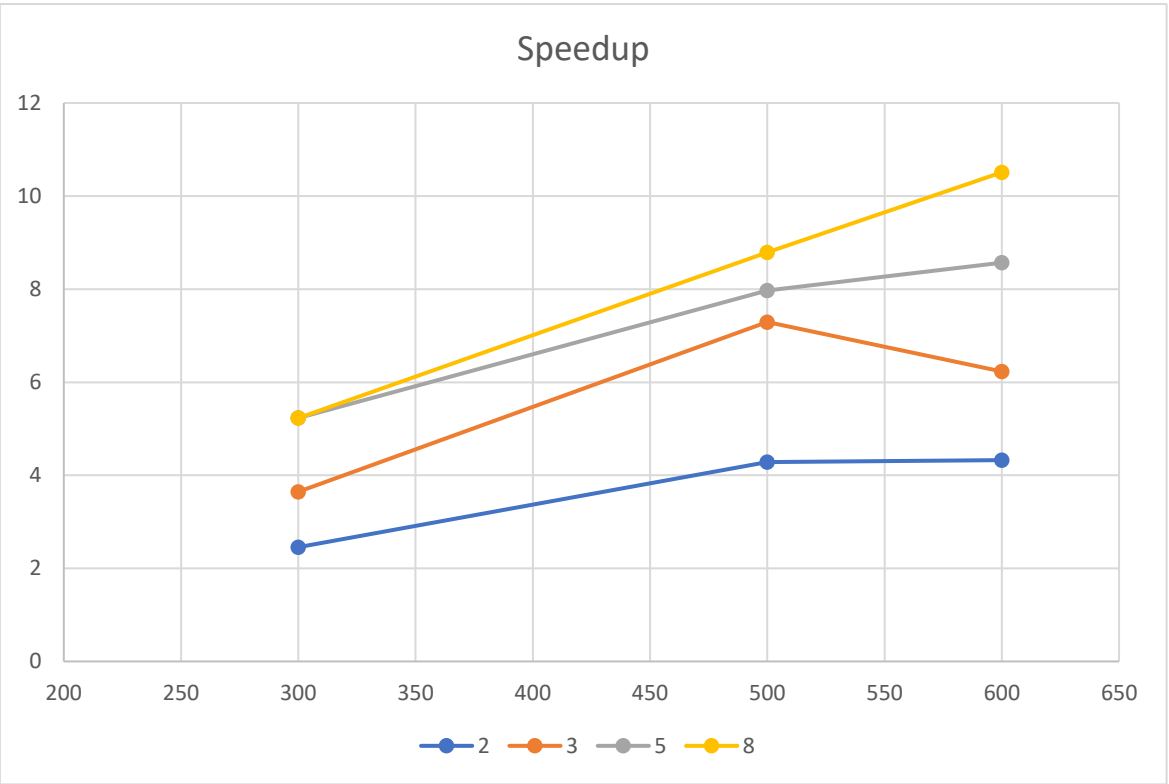
Таблиця швидкості алгоритму Фокса у мс

Size\Threads	2	3	5	8
300	49	33	23	23
500	160	94	86	78
600	311	216	157	128



Тепер порахуємо прискорення відносно послідовного алгоритму

Speedup	2	3	5	8
300	2,455	3,645303	5,230217	5,230217
500	4,2847	7,293106	7,971535	8,789128
600	4,325555	6,227998	8,568455	10,50975



З таблиці швидкості алгоритму Фокса можна побачити, що збільшення кількості потоків призводить до зменшення часу виконання. Це відбувається до певного моменту, після якого збільшення кількості потоків не призводить до подальшого зменшення часу виконання. Наприклад, для розміру матриці 300, час виконання стабілізується при 5 потоках.

Таблиця прискорення показує, наскільки швидше алгоритм Фокса виконується порівняно з послідовним алгоритмом. Знову ж таки, прискорення збільшується з кількістю потоків до певного моменту, після якого додаткові потоки не призводять до подальшого збільшення прискорення.

Отже, можна зробити висновок, що алгоритм Фокса ефективно використовує багатопоточність для прискорення обчислень, але існує певний поріг, після якого додаткові потоки не призводять до подальшого збільшення швидкості виконання.

Завдання 3

Клієнт-серверна система та розподілена система з рівноправними процесорами представляють два різних підходи до обчислень.

В клієнт-серверній системі, є один сервер, який обробляє запити від багатьох клієнтів. В контексті алгоритму Фокса, сервер може бути відповідальний за виконання обчислень, в той час як клієнти відправляють матриці для множення та отримують результати. Це може бути ефективним, якщо сервер має високу обчислювальну потужність, але може стати вузьким місцем, якщо є велика кількість клієнтів або великі матриці для обчислення.

В розподіленій системі з рівноправними процесорами, обчислення розподіляються між всіма процесорами в системі. Кожен процесор може виконувати частину обчислень, а потім результати можуть бути зібрані разом. В контексті алгоритму Фокса, це може означати, що кожен процесор виконує множення підматриць. Це може бути більш ефективним для великих матриць або великої кількості обчислень, оскільки обчислювальне навантаження розподіляється між багатьма процесорами.

В обох випадках, алгоритм Фокса може бути ефективно реалізований, але вибір між клієнт-серверною та розподіленою системою залежатиме від конкретних вимог до проекту, включаючи розмір матриць, кількість обчислень, що потрібно виконати, та доступні обчислювальні ресурси.

Висновок

У цій лабораторній роботі було виконано розробку веб-застосування клієнт-серверної архітектури, що реалізує алгоритм множення матриць на стороні сервера з використанням паралельних обчислень. Було розглянуто два варіанти реалізації.

Також було проведено дослідження швидкості виконання запиту користувача при різних обсягах даних. Результати дослідження демонструють, що збільшення кількості потоків призводить до зменшення часу виконання, але лише до певного порогу, після якого додаткові потоки не призводять до подальшого збільшення швидкості виконання.

Нарешті, було проведено порівняння реалізації алгоритму в клієнт-серверній системі та в розподіленій системі з рівноправними процесорами. Висновок з цього порівняння показує, що вибір між клієнт-серверною та розподіленою системою залежить від конкретних вимог до проекту, включаючи розмір матриць, кількість обчислень, що потрібно виконати, та доступні обчислювальні ресурси.

Посилання на код: https://github.com/valeriia-radzivilo/parallel_lab8