Звіт

«ПАРАЛЕЛЬНІ АЛГОРИТМИ

МАТРИЧНО-ВЕКТОРНОГО МНОЖЕННЯ»

З дисципліни «Кластерні розрахунки»

Студентки 5го курсу

Групи ПЗС-1

Нортман Юлії Олександрівни

Зміст

Послідовний алгоритм множення матриці на вектор

Теоретична частина

У першій частині лабораторної роботи було реалізовано послідовний алгоритм множення квадратної матриці n*n на вектор-стовпчик розмірності n. В результаті роботи отримали вектор-рядочок розмірності n, кожен i-й елемент c_i якого є результатом скалярного множення i-ї строки матриці a_i на вектор b.

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} * b_j \, , 1 \leq i \leq n$$

Отже для того, щоб отримати результуючий вектор c необхідно виконати n однотипних операцій множення строк матриці A на вектор b. Кожна така операція включає в себе поелементне множення елементів рядочка матриці A і елементів вектора b з їх подальшим сумуванням.

```
// Function for matrix-vector multiplication
private void resultCalculation() {
    int i, j; // Loop variables
    for (i=0; i<size; i++) {
        pResult[i] = 0;
        for (j=0; j<size; j++)
            pResult[i] += pMatrix[i*size+j]*pVector[j];
    }
}
```

1- Послідовне множення матриці на вектор

Реалізація програми

Програма реалізована мовою Java 11.

У програмі використовуються наступні змінні:

```
private int size; // розмірність

private double[] pMatrix; /*квадратна матриця A розмірністю size*size, яка
представлена одномірним масивом*/
private double[] pVector; //вектор b розмірністю size
private double[] pResult; //результуючий вектор c розмірністю size
```

Ініціалізація відбувається за допомогою генератора випадкових чисел. Числа вибираються з діапазону [0, ... 500).

```
// Function for random initialization of objects' elements
private void randomDataInitialization (int size) {
    Random random = new Random();
    pVector = random.doubles(size, 0, 500).toArray();
    pMatrix = random.doubles(size*size, 0, 500).toArray();
}
```

Проведення обчислювальних експериментів

Згідно з алгоритмом, для отримання результату необхідно провести n однотипних операцій множення рядочка матриці на вектор. Кожна з цих операцій в свою чергу включає в себе множення елементів строки матриці на елементи вектора, що в сумі дає n операцій. Після цього виконується сумування отриманих добутків (n-1 операція). Тобто загальна кількість операцій рахується за формулою $n * (n + n - 1) = n * (2n - 1)$. Знаючи час виконання однієї операції τ можна оцінити час роботи алгоритму за формулою:$T = N * \tau = \ n * (2n - 1) * \tau$.

Для визначення часу виконання однієї операції виберемо один з експериментів як зразок. Нехай це буде експеримент, у якому розмірність матриці складає 5.000 елементів. Кількість операцій N отримаємо за формулою $N = 5.000 * (2 * 5.000 - 1) = 49.995.000$. Час роботи алгоритму склав 34417 мкс, тоді час виконання однієї операції τ = 6,8840885e^{-4} мкс.

Для тестування роботи програми запустимо її з різними вхідними значеннями розмірностей і виміряємо час роботи алгоритму. Результати наведені у Таблиця 1.

Номер тесту	Розмір матриці	Час роботи (мкс)	Теоретичний час (мкс)
Тест №1	10	5	0,1
Тест №2	100	494	13
Тест №3	1000	5.476	1.376
Тест №4	2000	7.069	5.505
Тест №5	3000	13.339	12.389
Тест №6	4000	23.502	22.026
Тест №7	5000	34.951	34.417
Тест №8	6000	48.848	49.561
Тест №9	7000	70.789	67.459
Тест №10	8000	90.068	88.110
Тест №11	9000	113.986	111.516
Тест №12	10000	144.175	137.674

Таблиця 1 - Результати роботи послідовного алгоритму множення матриці на вектор

Паралельний алгоритм множення матриці на вектор

Теоретична частина

Для даної задачі характерним є повторення одних і тих самих обчислювальних операцій для різних елементів матриці. В такому випадку можливе розпаралелювання задачі між різними процесами. При розв’язанні даної задачі використовувалося стрічкове розбиття матриці, тобто кожному процесу виділяється неперервна підмножина строк матриці. Розбиття відбувається на

неперервній основі. Для розв'язання задачі кожному процесу потрібно мати підмножину строк матриці A та вектор b.

Реалізація програми

Для реалізації алгоритму використовувалась мова Java 11 і бібліотека Mpj Express.

Перед початком роботи програми необхідно ініціалізувати середовище виконання MPI за допомогою функції MPI.Init(args), у якості параметрів якій передаються аргументи командного рядка. Аналогічно перед завершенням програми для того, щоб коректно закрити середовище виконання необхідно викликати метод MPI.Finalize(). Для визначення кількості процесів та рангу (номеру) конкретного процесу використовуються функції MPI.COMM_WORLD.Size() та MPI.COMM_WORLD.Rank() відповідно.

У програмі використовуються наступні змінні:

```
private final int procNum; // Кількість процесів
private final int procRank; // Ранг поточного процесу

private double[] pMatrix; /* матриця розмірності size*size, представлена у
вигляді одномірного масиву*/
private double[] pVector; // Вектор розмірності size
private double[] pResult; // Результуючий вектор розмірності size
private final int size; // Розмірність

double[] pProcRows; // Підмножина строк матриці для процесу з рангом procRank
double[] pProcResult; // Частина результуючого вектора, яка обраховується
//поточним процесом
private int rowNum; // кількість рядочків вхідної матриці, призначені для
//обробки поточним процесом

private final int MASTER = 0; // Ранг головного процеса
```

Перед тим, як переходити безпосередньо до алгоритму множення необхідно розділити початкову матрицю між процесами при чому таким чином, щоб на кожен процес надійшов приблизно однаковий об'єм роботи. Для цього кількість строк матриці A, яка буде оброблятися процесом з рангом і буде обраховуватися за формулою:

$rowNum = \lfloor restRows/(procNum - i) \rfloor$, де restRows – кількість ще нерозподілених строк.

```
int restRows; // Number of rows, that haven't been distributed yet
int i; // Loop variable
restRows = size;
for (i=0; i<procRank; i++) {
    restRows = restRows - restRows / (procNum - i);
}
// Determine the number of matrix rows stored on each process
rowNum = restRows/(procNum-procRank);
```

Початкова матриця та вектор ініціалізуються лише головним процесом.

```
if (procRank == MASTER) {
    // Initial matrix exists only on the pivot process
    pMatrix = new double [size*size];
    pVector = new double [size];
    // Values of elements are defined only on the pivot process
    randomDataInitialization();
}
```

Після цього необхідно передати іншим процесам необхідні їм дані. Вектор передається повністю кожному процесу за допомогою функції Bcast(). У якості параметрів вона приймає об'єкт який передається (одномірний масив), зсув починаючи з якого потрібно передавати дані (в даному випадку 0), розмір даних які передаються (розмірність вектору), тип даних та ранг процеса, який виконує розсилку.

```
MPI.COMM_WORLD.Bcast(pVector, 0,size, MPI.DOUBLE, MASTER);
```

Як було зазначено кожен процес потребує лише певну кількість рядків початкової матриці, для того щоб запустити алгоритм. Для розподілення частин матриці використовується функція Scatterv(), якій потрібні два масиви. Елементи першого масиву зберігають в собі кількість елементів для кожного процесу, елементи другого – зсув елементів, призначених кожному процесу від початку масиву.

```
pSendNum = new int[procNum]; // the number of elements sent to the process
pSendInd = new int[procNum]; // the index of the first data element sent to
the process
int restRows = size; // Number of rows, that haven't been distributed yet

// Determine the disposition of the matrix rows for current process
rowNum = (size / procNum);
pSendNum[0] = rowNum * size;
pSendInd[0] = 0;
for (int i = 1; i < procNum; i++) {
    restRows -= rowNum;
    rowNum = restRows / (procNum - i);
    pSendNum[i] = rowNum * size;
    pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];
}
// Scatter the rows
MPI.COMM_WORLD.Scatterv(pMatrix, 0, pSendNum, pSendInd, MPI.DOUBLE,
        pProcRows, 0, pSendNum[procRank], MPI.DOUBLE, MASTER);
```

Аналогічно для збору даних використовується функція Allgatherv().

```
int i; // Loop variable
pReceiveNum = new int[procNum]; // Number of elements, that current process
sends
pReceiveInd = new int[procNum]; /* Index of the first element from current
process
in result vector */
int restRows=size; // Number of rows, that haven't been distributed yet

// Detrmine the disposition of the result vector block of current processor
pReceiveInd[0] = 0;
pReceiveNum[0] = size/procNum;
for (i=1; i<procNum; i++) {
    restRows -= pReceiveNum[i-1];
    pReceiveNum[i] = restRows/(procNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
}
MPI.COMM_WORLD.Allgatherv(pProcResult, 0, pReceiveNum[procRank], MPI.DOUBLE,
        pResult, 0, pReceiveNum, pReceiveInd, MPI.DOUBLE);
```

Проведення обчислювальних експериментів

Для тестування роботи програми запустимо її з різними вхідними значеннями розмірностей і виміряємо час роботи алгоритму. Результати наведено у Таблиця 2.

Розмір	Послідовний алгоритм	Паралельний алгоритм					
		2 процеси		4 процеси		8 процесів	
		Теоретичний час (мкс)	Час роботи	Теоретичний час (мкс)	Час роботи	Теоретичний час (мкс)	Час роботи
10	5	0,3	2464	0,4	3245	0,4	7216
100	494	34	643	38	1199	41	3460
1000	5.476	3493	6154	3843	11097	4192	16390
2000	7.069	13976	18882	15378	21181	16772	24211
3000	13.339	31449	47352	34605	67368	37742	61233
4000	23.502	55912	50778	61523	51353	67099	52146
5000	34.951	87366	87929	96133	101433	104846	109027
6000	48.848	125809	177427	138433	151584	150980	164589
7000	70.789	171242	147162	188426	162307	205504	157980
8000	90.068	223665	195676	246109	202029	268415	203583
9000	113.986	283078	310605	311484	327484	339716	361479
10000	144.175	349481	301773	384551	315063	419405	360141

Таблиця 2