**Міністерство освіти і науки України**
**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
**Факультет інформатики та обчислювальної техніки**
**Кафедра обчислювальної техніки**

# Лабораторна робота 5

**«Бібліотека OpenMP. Бар’єри, критичні секції»**

з дисципліни

«Програмне забезпечення високопродуктивних комп’ютерних систем»

Виконав
студент групи ІМ-13
Кравчук Ілля Володимирович

Перевірив:
доц. Корочкін О. В.

**Київ 2024**

# Завдання

Розробити паралельний алгоритм для рішення математичної задачі, яка буде виконуватися на комп'ютерній системі з чотирма процесорами та чотирма пристроями вводу-виводу. Описати алгоритм виконання кожного потоку програми, використовуючи засоби організації взаємодії потоків, такі як бар'єри та критичні секції бібліотеки OpenMP. Розробити паралельну програму на мові C++. Провести налагодження програми та перевірити правильність результатів обчислень.

Варіант 14

$Z = (B*X)*(d*E + R*(MZ*MR))*p$

Введення – виведення даних

1: MZ, E.

2: Z, R, p.

3: B, MR.

4: X, d.

**Етап 1**. Побудова паралельного математичного алгоритму.

1) $ai = (Bн*Xн);$ i = 1...P
2) $a = a + ai;$ CP: a;
3) $Zн = a * (d * Eн + R * (MZ * MRн)) * p;$ CP: a, R, MZ, d, p;

N - розмірність вектора/матриці.

P - кількість потоків, які виконують обчислення.

H = N / P

Захисту потребує CP скаляр "a" при перезаписі і копіюванні та CP скаляр "d" і "p" при копіюванні.

**Етап 2.** Розробка алгоритмів потоків.

| T1 | Точки синхронізації |
|---|---|
| 1. Ведення MZ, E. | |
| 2. *Сигнал* задачам T2, T3, T4 про введення MZ, E. | -- $S_{2-1}$, $S_{3-1}$, $S_{4-1}$ |
| 3. *Чекати* на введення даних в задачах T2, T3, T4 | -- $W_{2-1}$, $W_{3-1}$, $W_{4-1}$ |
| 4. Обчислення 1: a1 = (Bн * Xн) | |
| 5. Обчислення 2: a = a + a1 | -- КД1 |

6. *Сигнал* задачам T2, T3, T4 про завершення обчислення 2 -- $S_{2\text{-}2}$, $S_{3\text{-}2}$, $S_{4\text{-}2}$
7. *Чекати* на завершення обчислення 2 в задачах T2, T3, T4 -- $W_{2\text{-}2}$, $W_{3\text{-}2}$, $W_{4\text{-}2}$
8. *Копіювання* a1 = a -- КД2
9. *Копіювання* d1 = d -- КД3
10. *Копіювання* p1 = p -- КД4
11. Обчислення 3: Zн = a1 * (d1 * Ен + R * (MZ * MRн)) * p1
12. *Сигнал* задачі T2 про завершення обчислення 3 -- $S_{2\text{-}3}$

T2

1. Ведення R, p.
2. *Сигнал* задачам T1, T3, T4 про введення R, p. -- $S_{1\text{-}1}$, $S_{3\text{-}1}$, $S_{4\text{-}1}$
3. *Чекати* на введення даних в задачах T1, T3, T4 -- $W_{1\text{-}1}$, $W_{3\text{-}1}$, $W_{4\text{-}1}$
4. Обчислення 1: a2 = (Bн * Xн)
5. Обчислення 2: a = a + a2 -- КД1
6. *Сигнал* задачам T1, T3, T4 про завершення обчислення 2 -- $S_{1\text{-}2}$, $S_{3\text{-}2}$, $S_{4\text{-}2}$
7. *Чекати* на завершення обчислення 2 в задачах T1, T3, T4 -- $W_{1\text{-}2}$, $W_{3\text{-}2}$, $W_{4\text{-}2}$
8. *Копіювання* a2 = a -- КД2
9. *Копіювання* d2 = d -- КД3
10. *Копіювання* p2 = p -- КД4
11. Обчислення 3: Zн = a2 * (d2 * Ен + R * (MZ * MRн)) * p2
12. *Чекати* на завершення обчислення 3 в задачах T1, T3, T4 -- $W_{1\text{-}3}$, $W_{3\text{-}3}$, $W_{4\text{-}3}$
13. Виведення результату Z

T3

1. Ведення B, MR
2. *Сигнал* задачам T1, T2, T4 про введення B, MR -- $S_{1\text{-}1}$, $S_{2\text{-}1}$, $S_{4\text{-}1}$
3. *Чекати* на введення даних в задачах T1, T2, T4 -- $W_{1\text{-}1}$, $W_{2\text{-}1}$, $W_{4\text{-}1}$
4. Обчислення 1: a3 = (Bн * Xн)
5. Обчислення 2: a = a + a3 -- КД1
6. *Сигнал* задачам T1, T2, T4 про завершення обчислення 2 -- $S_{1\text{-}2}$, $S_{2\text{-}2}$, $S_{4\text{-}2}$
7. *Чекати* на завершення обчислення 2 в задачах T1, T2, T4 -- $W_{1\text{-}2}$, $W_{2\text{-}1}$, $W_{4\text{-}2}$
8. *Копіювання* a3 = a -- КД2
9. *Копіювання* d3 = d -- КД3
10. *Копіювання* p3 = p -- КД4
11. Обчислення 3: Zн = a3 * (d3 * Ен + R * (MZ * MRн)) * p3
12. *Сигнал* задачі T2 про завершення обчислення 3 -- $S_{2\text{-}3}$

T4

1. Ведення X, d.

2. *Сигнал* задачам T1, T2, T3 про введення X, d. -- $S_{1-1}$, $S_{2-1}$, $S_{3-1}$,

3. *Чекати* на введення даних в задачах T1, T2, T3 -- $W_{1-1}$, $W_{2-1}$, $W_{3-1}$

4. Обчислення 1: a4 = (Bн * Xн)

5. Обчислення 2: a = a + a4 -- КД1

6. *Сигнал* задачам T1, T2, T3 про завершення обчислення 2 -- $S_{1-2}$, $S_{2-2}$, $S_{3-2}$

7. *Чекати* на завершення обчислення 2 в задачах T1, T2, T3 -- $W_{1-2}$, $W_{2-1}$, $W_{3-2}$

8. *Копіювання* a4 = a -- КД2

9. *Копіювання* d4 = d -- КДЗ

10. *Копіювання* p4 = p -- КД4

11. Обчислення 3: Zн = a4 * (d4 * Eн + R * (MZ * MRн)) * p4

12. *Сигнал* задачі T2 про завершення обчислення 3 -- $S_{2-3}$

**Етап 3**. Розробка схеми взаємодії задач.

Бар'єри призначені для синхронізації введення, обчислення a, виведення(Рис.1).

CS1 – критична секція для захисту СР **a**, під час обчислення(Рис.1).

CS2 – критична секція для захисту СР **a**, під час копіювання(Рис.1).

CS3 – критична секція для захисту СР **d**, під час копіювання(Рис.1).

CS4 – критична секція для захисту СР **p**, під час копіювання(Рис.1).

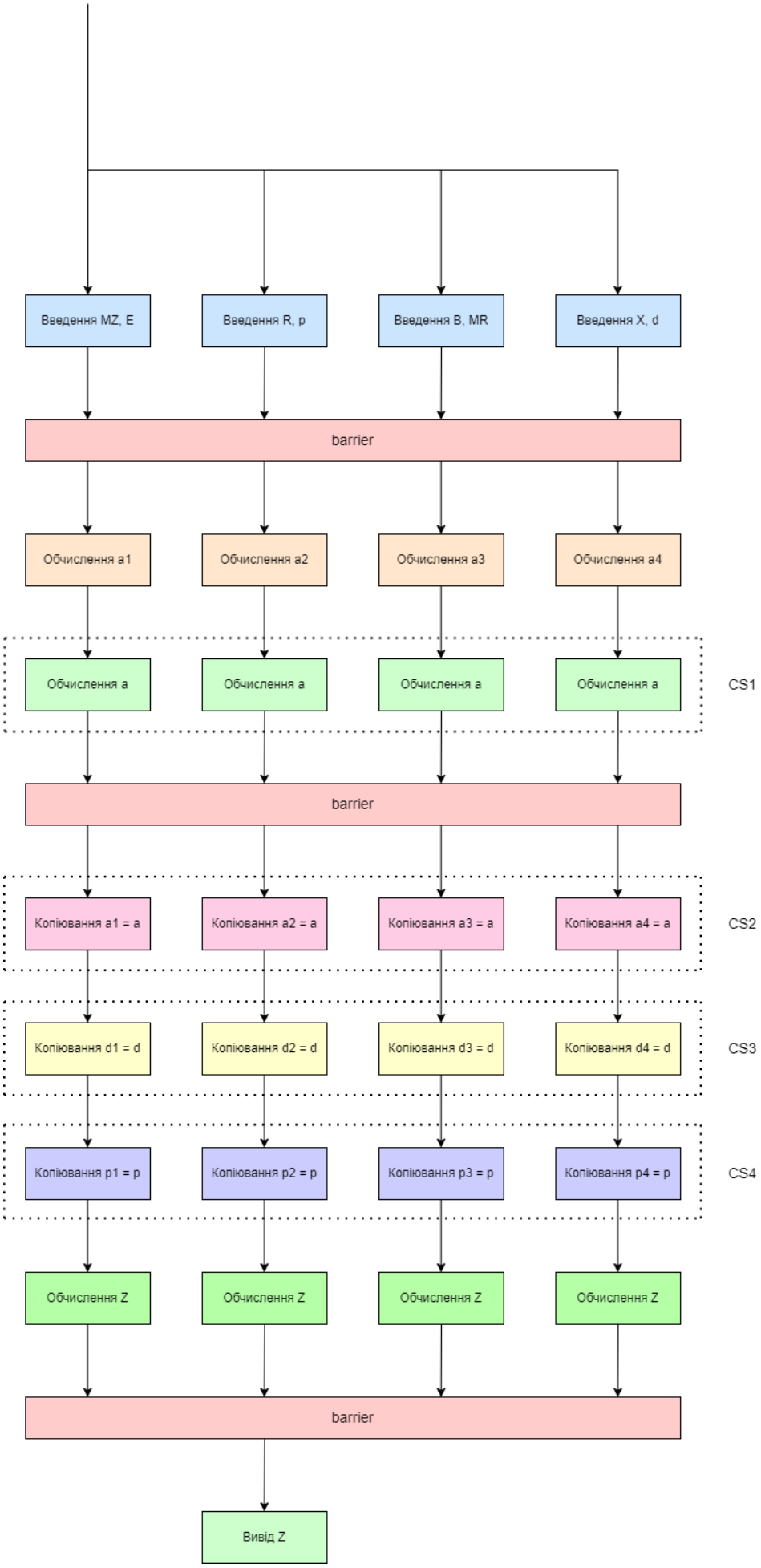


Рис1.Схема взаємодії задач

**Етап 4.** Розроблення програми.

## Lab_5.cpp (з pragma for)

```
// ПЗВПКС
// Лабораторна робота №5
// Варіант 14
// Z = (B*X)*(d*E + R*(MZ*MR))*p
// Кравчук Ілля Володимирович ІМ-13
// Дата 26.05.2024

#include <iostream>
#include <chrono>
#include <omp.h>

int generateScalar();
int* generateVector(int size);
int** generateMatrix(int size);
int scalarProduct(int* vec1, int* vec2, int startIdx, int endIdx);
int** extractSubMatrix(int** matrix, int section);
int* extractSubVector(int* vector, int section);
int** matrixMultiplication(int** mat1, int** mat2);
int* vectorMatrixMultiplication(int* vector, int** matrix);
int* scalarVectorMultiplication(int scalar, int* vector);
int* vectorAddition(int* vec1, int* vec2);
void computeTemp3(int* resultVec, int* temp3, int scalarD, int* vecE, int* vecR, int**
matMZ, int** matMR, int threadId);
void printResultVector(int* vector, int size);
void freeMemory();

const int N = 1000;
const int P = 4;
const int H = N / P;

int a = 0;
int d;
int p;
int* vecZ = new int[N];
int* vecE;
int* vecR;
int* vecB;
int* vecX;
int** matMZ;
int** matMR;

int main() {
    auto startTime = std::chrono::high_resolution_clock::now();
    int a_i;
    int d_i;
    int p_i;
    int threadId;

    omp_set_num_threads(P);

#pragma omp parallel num_threads(P) private(threadId, a_i, d_i, p_i) shared(a, d, p,
vecE, vecR, vecB, vecX, matMZ, matMR, vecZ)
    {
        threadId = omp_get_thread_num() + 1;

#pragma omp critical
        {
            std::cout << "Thread_" << threadId << " is started" << std::endl;
        }

        switch (threadId) {
        case 1: // T1
            matMZ = generateMatrix(N); //Введення MZ
```

```
            vecE = generateVector(N);  //Введення E
            break;
        case 2: // T2
            vecR = generateVector(N);  //Введення R
            p = generateScalar();  //Введення p
            break;
        case 3: // T3
            vecB = generateVector(N);  //Введення B
            matMR = generateMatrix(N);  //Введення MR
            break;
        case 4: // T4
            vecX = generateVector(N);  //Введення X
            d = generateScalar();  //Введення d
            break;
        }

        // Бар'єр для синхронізації по введенню
#pragma omp barrier

        // Обчислення 1: ai = Bh * Xh
        int startIdx;
        int endIdx;
        switch (threadId) {
        case 1: // T1
            startIdx = 0;
            endIdx = H;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T1
            break;
        case 2: // T2
            startIdx = H;
            endIdx = H * 2;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T2
            break;
        case 3: // T3
            startIdx = H * 2;
            endIdx = H * 3;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T3
            break;
        case 4: // T4
            startIdx = H * 3;
            endIdx = N;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T4
            break;
        }

        //Обчислення 2  a = a + ai   -- КД1
#pragma omp critical(CS)
        {
            a = a + a_i;
        }

        // Бар'єр для синхронізаціЇ обчислення 2
#pragma omp barrier

        // копіювання a_i = a    --КД2
#pragma omp critical(CS)
        {
            a_i = a;
        }

        // копіювання d_i = d    --КД3
#pragma omp critical(CS)
        {
            d_i = d;
```

```
        }

        // копіювання p_i = p    --КД4
#pragma omp critical(CS)
        {
            p_i = p;
        }

        int* temp3 = new int[H];

        computeTemp3(vecZ, temp3, d_i, vecE, vecR, matMZ, matMR, threadId);
        // Обчислення 3  vecZ = a_i * temp3 * p_i;
#pragma omp parallel for
        for (int i = 0; i < N; ++i) {
            vecZ[i] = a_i * temp3[i % H] * p_i;
        }

        delete[] temp3;

        // Бар'єр для синхронізаціЇ виведення Z
#pragma omp barrier

        if (threadId == 2) {
#pragma omp critical
            {
                //Вивід Z
                std::cout << "Z vector" << std::endl;
                printResultVector(vecZ, N);
                std::cout << std::endl;
            }
        }

#pragma omp critical
        {
            std::cout << "Thread_" << threadId << " is finished" << std::endl;
        }
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime);
    std::cout << "Program execution time: " << duration.count() << "ms" << std::endl;

    freeMemory();

    return 0;
}


int generateScalar() {
    return 1;
}

int* generateVector(int size) {
    int* res = new int[size];
    for (int i = 0; i < size; i++) {
        res[i] = 1;
    }
    return res;
}

int** generateMatrix(int size) {
    int** res = new int* [size];
    for (int i = 0; i < size; i++) {
        res[i] = new int[size];
        for (int j = 0; j < size; j++) {
            res[i][j] = 1;
        }
```

```
    }
    return res;
}

void printResultVector(int* vector, int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << vector[i] << " ";
    }
    std::cout << std::endl;
}

int scalarProduct(int* vec1, int* vec2, int startIdx, int endIdx) {
    int result = 0;
    for (int i = startIdx; i < endIdx; ++i) {
        result += vec1[i] * vec2[i];
    }
    return result;
}

int** extractSubMatrix(int** matrix, int section) {
    int** res = new int* [N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[H];
    }
    int startIdx = (section - 1) * H;
    int endIdx = startIdx + H;
    for (int j = startIdx; j < endIdx; ++j) {
        for (int i = 0; i < N; ++i) {
            res[i][j - startIdx] = matrix[i][j];
        }
    }
    return res;
}

int* extractSubVector(int* vector, int section) {
    int* res = new int[H];
    int startIdx = (section - 1) * H;
    int endIdx = startIdx + H;
    for (int i = startIdx; i < endIdx; ++i) {
        res[i - startIdx] = vector[i];
    }
    return res;
}

int** matrixMultiplication(int** mat1, int** mat2) {
    int** res = new int* [N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[N];
    }
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < H; ++j) {
            res[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                res[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
    return res;
}

int* vectorMatrixMultiplication(int* vector, int** matrix) {
    int* res = new int[H];
    for (int i = 0; i < H; i++) {
        int sum = 0;
        for (int j = 0; j < N; j++) {
            sum += matrix[j][i] * vector[j];
        }
        res[i] = sum;
```

```
    }
    return res;
}

int* scalarVectorMultiplication(int scalar, int* vector) {
    int* res = new int[N];
    for (int i = 0; i < N; i++) {
        res[i] = scalar * vector[i];
    }
    return res;
}

int* vectorAddition(int* vec1, int* vec2) {
    int* res = new int[H];
    for (int i = 0; i < H; i++) {
        res[i] = vec1[i] + vec2[i];
    }
    return res;
}

// Обчислення  temp3 = (d * E + R * (MZ * MR));
void computeTemp3(int* resultVec,  int* temp3, int scalarD, int* vecE, int* vecR,
int** matMZ, int** matMR, int threadId) {
    int** subMatMR = extractSubMatrix(matMR, threadId);
    int* subVecE = extractSubVector(vecE, threadId);

    int* temp2 = vectorMatrixMultiplication(vecR, matrixMultiplication(matMZ,
subMatMR));

    int* temp1 = scalarVectorMultiplication(scalarD, subVecE);
    int* temp3Part = vectorAddition(temp1, temp2);

    for (int i = 0; i < H; i++) {;
        temp3[i] = temp3Part[i];
    }

    for (int i = 0; i < N; i++) {
        delete[] subMatMR[i];
    }
    delete[] subMatMR;
    delete[] subVecE;
    delete[] temp1;
    delete[] temp2;
    delete[] temp3Part;
}

void freeMemory() {
    for (int i = 0; i < N; i++) {
        delete[] matMZ[i];
        delete[] matMR[i];
    }
    delete[] matMZ;
    delete[] matMR;
    delete[] vecB;
    delete[] vecE;
    delete[] vecR;
    delete[] vecX;
}
```

Скріншот виконання програми при N = 16 (Рис.2).

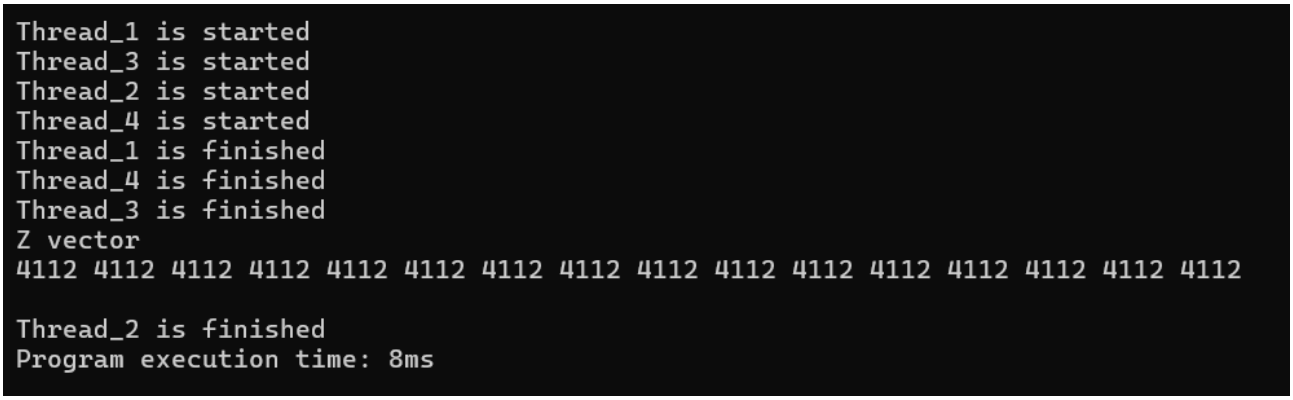


Рис.2 Скріншот виконання програми з використанням **pragma for**

## Lab_5.cpp (без pragma for)

```
// ПЗВПКС
// Лабораторна робота №5
// Варіант 14
// Z = (B*X)*(d*E + R*(MZ*MR))*p
// Кравчук Ілля Володимирович ІМ-13
// Дата 26.05.2024

#include <iostream>
#include <chrono>
#include <omp.h>

int generateScalar();
int* generateVector(int size);
int** generateMatrix(int size);
int scalarProduct(int* vec1, int* vec2, int startIdx, int endIdx);
int** extractSubMatrix(int** matrix, int section);
int* extractSubVector(int* vector, int section);
int** matrixMultiplication(int** mat1, int** mat2);
int* vectorMatrixMultiplication(int* vector, int** matrix);
int* scalarVectorMultiplication(int scalar, int* vector);
int* vectorAddition(int* vec1, int* vec2);
void mergeSubVector(const int* subVec, int* resultVec, int section);
void computeStep3(int* resultVec, int scalarA, int scalarD, int* vecE, int* vecR,
int** matMZ, int** matMR, int scalarP, int threadId);
void printResultVector(int* vector, int size);
void freeMemory();

const int N = 16;
const int P = 4;
const int H = N / P;

int a = 0;
int d;
int p;
int* vecZ = new int[N];
int* vecE;
int* vecR;
int* vecB;
int* vecX;
int** matMZ;
int** matMR;

int main() {
    auto startTime = std::chrono::high_resolution_clock::now();
    int a_i;
```

```
    int d_i;
    int p_i;
    int threadId;



    omp_set_num_threads(P);

#pragma omp parallel num_threads(P) private(threadId, a_i, d_i, p_i) shared(a, d, p,
vecE, vecR, vecB, vecX, matMZ, matMR, vecZ)
    {
        threadId = omp_get_thread_num() + 1;

#pragma omp critical
        {
            std::cout << "Thread_" << threadId << " is started" << std::endl;
        }

        switch (threadId) {
        case 1: // T1
            matMZ = generateMatrix(N); //Введення MZ
            vecE = generateVector(N);  //Введення E
            break;
        case 2: // T2
            vecR = generateVector(N);  //Введення R
            p = generateScalar();  //Введення p
            break;
        case 3: // T3
            vecB = generateVector(N);  //Введення B
            matMR = generateMatrix(N);  //Введення MR
            break;
        case 4: // T4
            vecX = generateVector(N);  //Введення X
            d = generateScalar();  //Введення d
            break;
        }

        // Бар'єр для синхронізації по введенню
#pragma omp barrier

        // Обчислення 1: ai = Bh * Xh
        int startIdx;
        int endIdx;
        switch (threadId) {
        case 1: // T1
            startIdx = 0;
            endIdx = H;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T1
            break;
        case 2: // T2
            startIdx = H;
            endIdx = H * 2;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T2
            break;
        case 3: // T3
            startIdx = H * 2;
            endIdx = H * 3;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T3
            break;
        case 4: // T4
            startIdx = H * 3;
            endIdx = N;
            a_i = scalarProduct(vecB, vecX, startIdx, endIdx); //Обчислення скалярного
добутку в T4
            break;
```

```
        }

        //Обчислення 2  a = a + ai   -- КД1
#pragma omp critical(CS)
        {
            a = a + a_i;
        }

        // Бар'єр для синхронізації обчислення 2
#pragma omp barrier

        // копіювання a_i = a    --КД2
#pragma omp critical(CS)
        {
            a_i = a;
        }

        // копіювання d_i = d    --КД3
#pragma omp critical(CS)
        {
            d_i = d;
        }

        // копіювання p_i = p    --КД4
#pragma omp critical(CS)
        {
            p_i = p;
        }

        // Обчислення 3  vecZ = a_i * (d_i * vecE + vecR * (matMZ * matMR)) * p_i;
        computeStep3(vecZ, a_i, d_i, vecE, vecR, matMZ, matMR, p_i, threadId);

        // Бар'єр для синхронізації виведення Z
#pragma omp barrier

        if (threadId == 2) {
#pragma omp critical
            {
                //Вивід Z
                std::cout << "Z vector" << std::endl;
                printResultVector(vecZ, N);
                std::cout << std::endl;
            }
        }

#pragma omp critical
        {
            std::cout << "Thread_" << threadId << " is finished" << std::endl;
        }
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime);
    std::cout << "Program execution time: " << duration.count() << "ms" << std::endl;


    freeMemory();

    return 0;
}

int generateScalar() {
    return 1;
}

int* generateVector(int size) {
    int* res = new int[size];
```

```
    for (int i = 0; i < size; i++) {
        res[i] = 1;
    }
    return res;
}

int** generateMatrix(int size) {
    int** res = new int* [size];
    for (int i = 0; i < size; i++) {
        res[i] = new int[size];
        for (int j = 0; j < size; j++) {
            res[i][j] = 1;
        }
    }
    return res;
}

void printResultVector(int* vector, int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << vector[i] << " ";
    }
    std::cout << std::endl;
}

int scalarProduct(int* vec1, int* vec2, int startIdx, int endIdx) {
    int result = 0;
    for (int i = startIdx; i < endIdx; ++i) {
        result += vec1[i] * vec2[i];
    }
    return result;
}

int** extractSubMatrix(int** matrix, int section) {
    int** res = new int* [N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[H];
    }
    int startIdx = (section - 1) * H;
    int endIdx = startIdx + H;
    for (int j = startIdx; j < endIdx; ++j) {
        for (int i = 0; i < N; ++i) {
            res[i][j - startIdx] = matrix[i][j];
        }
    }
    return res;
}

int* extractSubVector(int* vector, int section) {
    int* res = new int[H];
    int startIdx = (section - 1) * H;
    int endIdx = startIdx + H;
    for (int i = startIdx; i < endIdx; ++i) {
        res[i - startIdx] = vector[i];
    }
    return res;
}

int** matrixMultiplication(int** mat1, int** mat2) {
    int** res = new int* [N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[N];
    }
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < H; ++j) {
            res[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                res[i][j] += mat1[i][k] * mat2[k][j];
            }
```

```
        }
    }
    return res;
}

int* vectorMatrixMultiplication(int* vector, int** matrix) {
    int* res = new int[H];
    for (int i = 0; i < H; i++) {
        int sum = 0;
        for (int j = 0; j < N; j++) {
            sum += matrix[j][i] * vector[j];
        }
        res[i] = sum;
    }
    return res;
}

int* scalarVectorMultiplication(int scalar, int* vector) {
    int* res = new int[N];
    for (int i = 0; i < N; i++) {
        res[i] = scalar * vector[i];
    }
    return res;
}

int* vectorAddition(int* vec1, int* vec2) {
    int* res = new int[H];
    for (int i = 0; i < H; i++) {
        res[i] = vec1[i] + vec2[i];
    }
    return res;
}

void mergeSubVector(const int* subVec, int* resultVec, int section) {
    int startIdx = (section - 1) * H;
    for (int i = 0; i < H; i++) {
        resultVec[startIdx + i] = subVec[i];
    }
}

// Обчислення 3  Z = a * (d * E + R * (MZ * MR)) * p;
void computeStep3(int* resultVec, int scalarA, int scalarD, int* vecE, int* vecR,
int** matMZ, int** matMR, int scalarP, int threadId) {
    int** subMatMR = extractSubMatrix(matMR, threadId);
    int* subVecE = extractSubVector(vecE, threadId);
    int* temp1 = scalarVectorMultiplication(scalarD, subVecE);
    int* temp2 = vectorMatrixMultiplication(vecR, matrixMultiplication(matMZ,
subMatMR));
    int* temp3 = vectorAddition(temp1, temp2);
    int* temp4 = scalarVectorMultiplication(scalarA, temp3);
    int* subVecZ = scalarVectorMultiplication(scalarP, temp4);

    mergeSubVector(subVecZ, resultVec, threadId);

    for (int i = 0; i < N; i++) {
        delete[] subMatMR[i];
    }
    delete[] subMatMR;
    delete[] subVecE;
    delete[] temp1;
    delete[] temp2;
    delete[] temp3;
    delete[] temp4;
    delete[] subVecZ;
}

void freeMemory() {
    for (int i = 0; i < N; i++) {
```

```
        delete[] matMZ[i];
        delete[] matMR[i];
    }
    delete[] matMZ;
    delete[] matMR;
    delete[] vecB;
    delete[] vecE;
    delete[] vecR;
    delete[] vecX;
}
```

Скріншот виконання програми при N = 16 (Рис.3).

```
Thread_1 is started
Thread_2 is started
Thread_3 is started
Thread_4 is started
Thread_4 is finished
Z vector
4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112 4112

Thread_2 is finished
Thread_1 is finished
Thread_3 is finished
Program execution time: 10ms
```

Рис.3 Скріншот виконання програми

## Тестування програми

В ноутбуці є 14 ядер і 20 логічних процесорів(Рис.4).


Рис.4 Характеристики процесора

N = 1000.

**Програма без використання pragma for.**

Час на одному ядрі **5620** ms.(Рис.5)

```
01000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 100
000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 10000
0 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001
1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 100000100
00001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000
001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 10
1000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000
00 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000 1000001000

Thread_2 is finished
Program execution time: 5620ms
```

Рис.5 Час виконання програми на одному ядрі.

Час на чотирьох ядрах **2867** ms. (Рис.6)

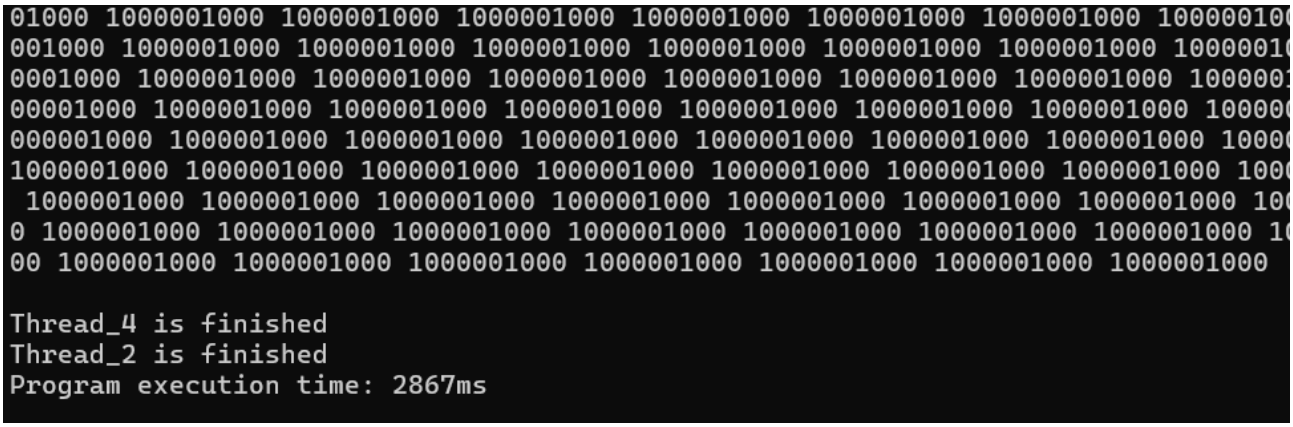


Рис.6 Час виконання на чотирьох ядрах

Кп = 5620/2867 = 1,96

**Програма з використання pragma for.**

Час на одному ядрі **5176** ms.(Рис.5)

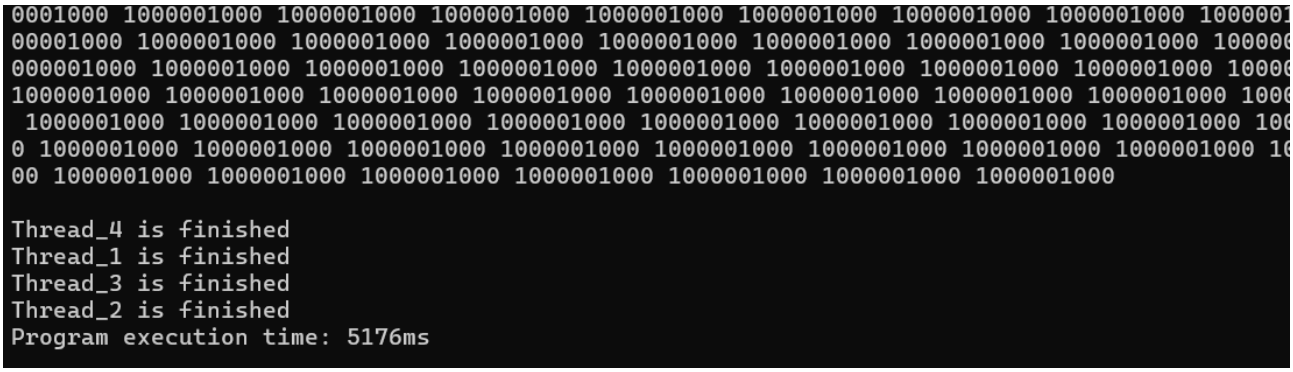


Рис.5 Час виконання програми на одному ядрі.

Час на чотирьох ядрах **2692** ms. (Рис.6)

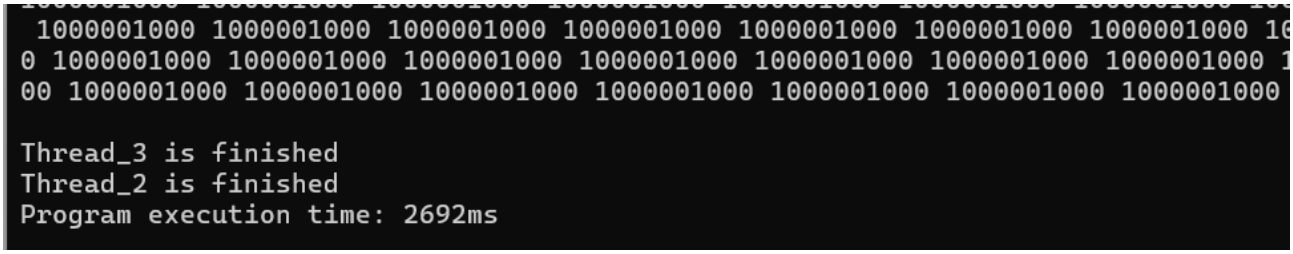


Рис.6 Час виконання на чотирьох ядрах

Кп = **5176** /**2692** = 1,92

## Висновок

1.Для створення багатопотокової програми використовувалась бібліотека OpenMP у мові C++, що дозволяє керувати потоками програми. Для створення потоків використовується директива #pragma omp parallel. Для задання локальних змінних застосовується ключове слово private, яке забезпечує

створення копії змінних для кожного потоку. Кількість потоків визначається за допомогою функції omp_set_num_threads(P), а для отримання номера потоку використовується функція omp_get_thread_num(). Для визначення критичних секцій застосовується директива #pragma omp critical, яка дозволяє виконувати код лише одним потоком одночасно. Встановлення бар'єрів, що синхронізують потоки, здійснюється за допомогою директиви #pragma omp barrier. Для автоматичного розподілу ітерацій циклу паралельно між потоками використовується директива #pragma omp parallel for.

2.Розроблено паралельний математичний алгоритм, який розділяє математичний вираз на підзадачі для виконання в паралельній системі. У цьому алгоритмі спільними ресурсами є: a, R, MZ, d, p;  .

3.Розроблено алгоритм для кожного потоку, у якому визначено точки синхронізації. Ці точки включають синхронізацію введення даних, виконання обчислення і виведення результату. Крім того, визначено чотири завдання взаємного виключення, пов'язані з перезаписом або копіюванням спільних ресурсів.

4.Побудовано структурну схему взаємодії алгоритмів, для якої обрано такі засоби синхронізації:

- Бар'єри: забезпечують синхронізацію взаємодії потоків.
- Критичні секції: захищають спільні ресурси.

5. Проведено тестування, під час якого була підтверджена ефективність багатопотокової програми з коефіцієнтом прискорення 1,92 при використанні #pragma omp parallel for та 1,96 без його використання. Це свідчить про те, що застосування #pragma omp parallel for не є дуже суттєвим, оскільки обидві програми працюють майже однаково.