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

з дисципліни
«Програмне забезпечення високопродуктивних комп’ютерних систем»

на тему

«Бібліотека OpenMP. Бар’єри, критичні секції»

Виконав
Студент групи ІМ-13
Котенко Ярослав Олегович

Перевірив
доц. Корочкін О.В.

Київ 2024

Розробити паралельну програму для обчислення в паралельної системі (ПКС СП) функції :

Варіант: 9

A= min(Z)*(B*MV) +e*X*(MM*MC)

Введення – виведення даних			
1	2	3	4
MV, MC	e, MM	A, Z	B, X

Мова програмування: C++

Засоби організації взаємодії процесів : бар'єри, критичні секції OpenMP.

Виконання лабораторної роботи

Етап 1. Побудова паралельного математичного алгоритму.

A = min(Z)*(B*MV) +e*X*(MM*MC)

1. ai = min(Zн)
2. a = min(a, ai) CP: a
3. Aн = a*(B*MVн) + e*X*(MM*MCн) CP: B, X, MM, a, e

N - розмірність вектора/матриці.

P - кількість потоків, які виконують обчислення.

H = N / P

Етап 2. Розробка алгоритмів потоків

Задача T1	**Точки синхронізації**
1. Введення MV, MC	
2. **Сигнал** задачі T2, T3, T4 про введення MB, MC	-- **S_{2-1}, S_{3-1}, S_{4-1}**
3. **Чекати** на введення даних у потоці T2, T3, T4	-- **W_{2-1}, W_{3-1}, W_{4-1}**
4. Обчислення 1: a1 = min(Zн)	
5. Обчислення 2: a = min(a, a1)	-- КД1
6. **Сигнал** T2, T3, T4 про завершення обчислення a	-- **S_{2-2}, S_{3-2}, S_{4-2}**
7. **Чекати** на завершення обчислень a у потоках T2, T3, T4	-- **W_{2-2}, W_{3-2}, W_{4-2}**
8. **Копія** e1 = e	-- КД2

9. **Копія** a1 = a -- КД3
10. Обчислення 3: Ан = a1*(B*MVн) + e1*X*(MM*MCн)
11. **Сигнал** про завершення обчислень Ан потоку Т3 -- S_{3-3}

Задача Т2

1. Введення e, MM
2. **Сигнал** задачі Т1, Т3, Т4 про введення e, MM -- S_{1-1}, S_{3-1}, S_{4-1}
3. **Чекати** на введення даних у потоках Т1, Т3, Т4 -- W_{1-1}, W_{3-1}, W_{4-1}
4. Обчислення 1: a2 = min(Zн)
5. Обчислення 2: a = min(a, a2) -- КД1
6. **Сигнал** Т1, Т3, Т4 про завершення обчислення a -- S_{1-2}, S_{3-2}, S_{4-2}
7. **Чекати** на завершення обчислень a у потоках Т1, Т3, Т4 -- W_{1-2}, W_{3-2}, W_{4-2}
8. **Копія** e2 = e -- КД2
9. **Копія** a2 = a -- КД3
10. Обчислення 3: Ан = a2*(B*MVн) + e2*X*(MM*MCн)
11. **Сигнал** про завершення обчислень Ан потоку Т3 -- S_{3-3}

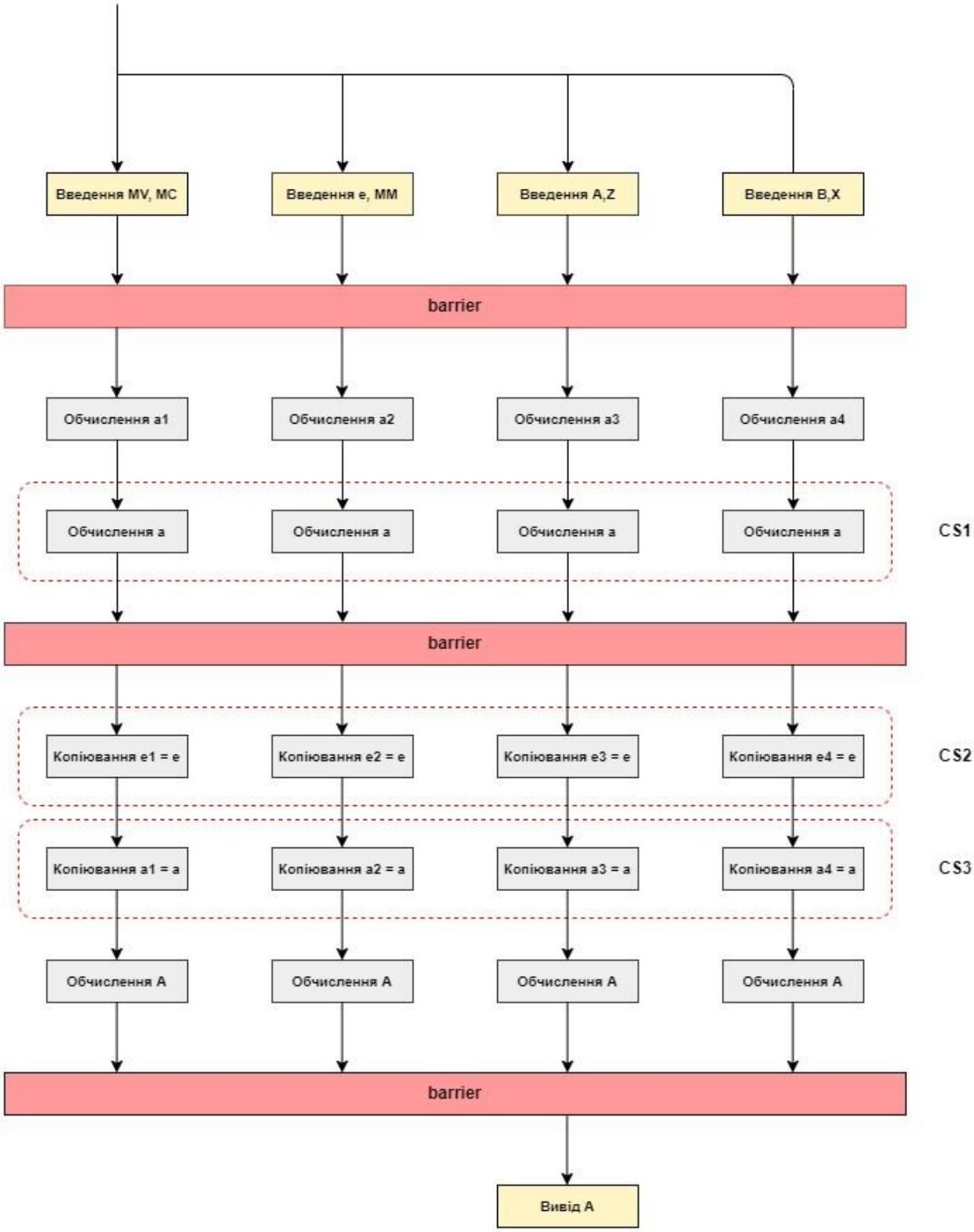
Задача Т3

1. Введення Z
2. **Сигнал** задачі Т1, Т2, Т4 про введення Z -- S_{1-1}, S_{2-1}, S_{4-1}
3. **Чекати** на введення даних у потоках Т1, Т2, Т4 -- W_{1-1}, W_{2-1}, W_{4-1}
4. Обчислення 1: a3 = min(Zн)
5. Обчислення 2: a = min(a, a3) -- КД1
6. **Сигнал** Т1, Т2, Т4 про завершення обчислення a -- S_{1-2}, S_{2-2}, S_{4-2}
7. **Чекати** на завершення обчислень a у потоках Т1, Т2, Т4 -- W_{1-2}, W_{2-2}, W_{4-2}
8. **Копія** e3 = e -- КД2
9. **Копія** a3 = a -- КД3
10. Обчислення 3: Ан = a3*(B*MVн) + e3*X*(MM*MCн)
11. **Чекати** на завершення обчислень Ан у потоках Т1, Т2, Т4 -- W_{1-3}, W_{2-3}, W_{4-3}
12. Виведення результату A

Задача Т4

1. Введення B, X
2. **Сигнал** задачі Т1, Т2, Т3 про введення B, X -- S_{1-1}, S_{2-1}, S_{3-1}
3. **Чекати** на введення даних у потоці Т1, Т2, Т3 -- W_{1-1}, W_{2-1}, W_{3-1}
4. Обчислення 1: a4 = min(Zн)
5. Обчислення 2: a = min(a, a4) -- КД1
6. **Сигнал** Т1, Т2, Т3 про завершення обчислення a -- S_{1-2}, S_{2-2}, S_{3-2}
7. **Чекати** на завершення обчислень a у потоках Т1, Т2, Т3 -- W_{1-2}, W_{2-2}, W_{3-2}
8. **Копія** e4 = e -- КД2
9. **Копія** a4 = a -- КД3
10. Обчислення 3: Ан = a4*(B*MVн) + e4*X*(MM*MCн)
11. **Сигнал** про завершення обчислень Ан потоку Т3 -- S_{3-3}

Етап 3. Розробка схеми взаємодії задач



Бар’єри призначені для для синхронізації введення, обчислення **a** та виведення фінального результату **A**.

CS1 – призначена для захисту під час обчислення спільного ресурсу **a**.

CS2 – призначена для захисту під час копіювання спільного ресурсу **e**.

CS3 – призначена для захисту під час копіювання спільного ресурсу **a**.

Етап 4. Розроблення програми.

Програма з використання конструкції pragma for

Lab5_pr.cpp

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

int fillScalar();
int* fillVector(int N);
int** fillMatrix(int N);
int findMinInVector(int* vector, int N);
int** getSubMatrix(int** matrix, int partOfMtrx);
int** multiplyMatrix(int** first_matrix, int** second_matrix);
int* multiplyVectorAndMatrix(int* vector, int** matrix);
int* multiplyScalarAndVector(int scalar, int* vector);
int* addVectors(int* first_vector, int* second_vector);
void combineVector(const int* vector, int* result, int partOfVctr);
void calculateStep3(int** MV, int** MM, int** MC, int* Z, int* B, int* X, int a, int e, int* A, int tId);
void printFinalVectorA(int* vector, int N);
void clearMemory();

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

int a = INT_MAX;
int e;
int* A;
int* B;
int* X;
int* Z;
int** MV;
int** MM;
int** MC;

int main()
{
   auto start_time = std::chrono::high_resolution_clock::now();
   int a_i;
   int e_i;
   int tId;

   omp_set_num_threads(P);

   #pragma omp parallel num_threads(P) private(tId, a_i,e_i) shared(a, e, B, Z, MC, MM, MV)
   {
      tId = omp_get_thread_num() + 1;

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

      switch (tId)
      {
```

```
case 1: // задача T1
   // Введення MV, MC
   MV = fillMatrix(N);
   MC = fillMatrix(N);
   break;
case 2: // задача T2
   // Введення e, MM
   e = fillScalar();
   MM = fillMatrix(N);
   break;
case 3: // задача T3
   // Введення A, Z
   A = fillVector(N);
   Z = fillVector(N);
   break;
case 4: // задача T4
   // Введення B, X
   B = fillVector(N);
   X = fillVector(N);
   break;
}

// Бар'єр для синхронізації введення
#pragma omp barrier

// Обчислення 1: ai = min(Zh)
a_i = findMinInVector(Z, N); // В методі знаходиться конструкція #pragma for

// КД1. Обчислення 2: a = min(a, ai)
#pragma omp critical(CS)
{
   if (a_i < a) {
      a = a_i;
   }
}

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

// КД2. Копія ei = e
#pragma omp critical(CS)
{
   e_i = e;
}

// КД3. Копія ai = a
#pragma omp critical(CS)
{
   a_i = a;
}

// Обчислення 3: Ан = ai * (B * MVн) + ei * X * (MM * MCн)
calculateStep3(MV, MM, MC, Z, B, X, a, e, A, tId);

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

if (tId == 3) {
#pragma omp critical
   {
      // Виведення фінального результату A в задачі T3
      std::cout << "A: [ ";
      printFinalVectorA(A, N);
      std::cout << "]" << std::endl;
   }
}
```

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

    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout << "Time: " << duration.count() << " ms" << std::endl;

    clearMemory();

    return 0;
}

// Метод для заповнення скаляру
int fillScalar() {
    return 1;
}

// Метод для заповнення вектора
int* fillVector(int N) {
    int* res = new int[N];
    for (int i = 0; i < N; i++) {
        res[i] = 1;
    }
    return res;
}

// Метод для заповнення матриці
int** fillMatrix(int N) {
    int** res = new int*[N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[N];
        for (int j = 0; j < N; j++) {
            res[i][j] = 1;
        }
    }
    return res;
}

// Метод для виведення фільного результату
void printFinalVectorA(int* vector, int N) {
    for (int i = 0; i < N; ++i) {
        std::cout << vector[i] << " ";
    }
}

// Метод для знаходження мінімуму вектора. Кострукція #pragma for
int findMinInVector(int* vector, int N) {
    int min = vector[0];
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        if (min > vector[i]) {
            min = vector[i];
        }
    }
    return min;
}

// Метод для отримання підматриці
int** getSubMatrix(int** matrix, int partOfMtrx) {
    int** res = new int* [N];

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

```
        res[i] = new int[H];
    }

    int startPos = (partOfMtrx - 1) * H;
    int endPos = startPos + H;

    for (int j = startPos; j < endPos; ++j) {
        for (int i = 0; i < N; ++i) {
            res[i][j - startPos] = matrix[i][j];
        }
    }

    return res;
}

// Метод для множення матриць
int** multiplyMatrix(int** first_matrix, int** second_matrix) {
    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] += first_matrix[i][k] * second_matrix[k][j];
            }
        }
    }

    return res;
}

// Метод для множення вектора на матрицю
int* multiplyVectorAndMatrix(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;
}

// Метод для множення скаляру на вектор
int* multiplyScalarAndVector(int scalar, int* vector) {
    int* res = new int[N];

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

    return res;
}

// Метод для додавання векторів
int* addVectors(int* first_vector, int* second_vector) {
    int* res = new int[H];

    for (int i = 0; i < H; i++) {
```

```
        res[i] = first_vector[i] + second_vector[i];
    }

    return res;
}

// Метод для об'єднання підвекторів в один вектор
void combineVector(const int* vector, int* result, int partOfVctr) {
    int startPos = (partOfVctr - 1) * H;

    for (int i = 0; i < H; i++) {
        result[startPos + i] = vector[i];
    }
}

// Метод для обчислення кроку 3
void calculateStep3(int** MV, int** MM, int** MC, int* Z, int* B, int* X, int a, int e, int* A, int tId) {
    int** MV_H = getSubMatrix(MV, tId);
    int** MC_H = getSubMatrix(MC, tId);

    int* Ah = addVectors(
        multiplyScalarAndVector(
            a,
            multiplyVectorAndMatrix(B, MV_H)
        ),
        multiplyScalarAndVector(
            e,
            multiplyVectorAndMatrix(
                X,
                multiplyMatrix(MM, MC_H)
            )
        )
    );
    combineVector(Ah, A, tId);
    for (int i = 0; i < N; i++) {
        delete[] MV_H[i];
        delete[] MC_H[i];
    }
    delete[] MV_H;
    delete[] MC_H;
}

// Метод для очищення пам'яті
void clearMemory() {
    for (int i = 0; i < N; i++) {
        delete[] MV[i];
        delete[] MC[i];
        delete[] MM[i];
    }
    delete[] MV;
    delete[] MC;
    delete[] MM;
    delete[] A;
    delete[] B;
    delete[] X;
    delete[] Z;
}
```

Результат виконання програми для N = 16.

```
Thread_1 is started
Thread_2 is started
Thread_4 is started
Thread_3 is started
Thread_1 is finished
Thread_4 is finished
Thread_2 is finished
A: [ 272 272 272 272 272 272 272 272 272 272 272 272 272 272 272 272 ]
Thread_3 is finished
Time: 8 ms
```

Програма без використання конструкції pragma for

Lab5.cpp

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

int fillScalar();
int* fillVector(int N);
int** fillMatrix(int N);
int findMinInSubVector(int* vector, int start, int end);
int** getSubMatrix(int** matrix, int partOfMtrx);
int** multiplyMatrix(int** first_matrix, int** second_matrix);
int* multiplyVectorAndMatrix(int* vector, int** matrix);
int* multiplyScalarAndVector(int scalar, int* vector);
int* addVectors(int* first_vector, int* second_vector);
void combineVector(const int* vector, int* result, int partOfVctr);
void calculateStep3(int** MV, int** MM, int** MC, int* Z, int* B, int* X, int a, int e, int* A, int tId);
void printFinalVectorA(int* vector, int N);
void clearMemory();

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

int a = INT_MAX;
int e;
int* A;
int* B;
int* X;
int* Z;
int** MV;
int** MM;
int** MC;

int main()
{
    auto start_time = std::chrono::high_resolution_clock::now();
    int a_i;
    int e_i;
    int tId;

    omp_set_num_threads(P);

    #pragma omp parallel num_threads(P) private(tId, a_i,e_i) shared(a, e, B, Z, MC, MM, MV)
    {
```

```
tId = omp_get_thread_num() + 1;

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

switch (tId)
{
case 1: // задача T1
   // Введення MV, MC
   MV = fillMatrix(N);
   MC = fillMatrix(N);
   break;
case 2: // задача T2
   // Введення e, MM
   e = fillScalar();
   MM = fillMatrix(N);
   break;
case 3: // задача T3
   // Введення A, Z
   A = fillVector(N);
   Z = fillVector(N);
   break;
case 4: // задача T4
   // Введення B, X
   B = fillVector(N);
   X = fillVector(N);
   break;
}

// Бар'єр для синхронізації введення
#pragma omp barrier

// Обчислення 1: ai = min(Zh)


int startPos;
int endPos;
switch (tId)
{
case 1: // задача T1
   startPos = 0;
   endPos = H;
   a_i = findMinInSubVector(Z, startPos, endPos); // знаходження мінімуму в першому підвекторі
   break;
case 2: // задача T2
   startPos = H;
   endPos = H * 2;
   a_i = findMinInSubVector(Z, startPos, endPos); // знаходження мінімуму в другому підвекторі
   break;
case 3: // задача T3
   startPos = H * 2;
   endPos = H * 3;
   a_i = findMinInSubVector(Z, startPos, endPos); // знаходження мінімуму в третьому підвекторі
   break;
case 4: // задача T4
   startPos = H * 3;
   endPos = N;
   a_i = findMinInSubVector(Z, startPos, endPos); // знаходження мінімуму в четвертому підвекторі
   break;
}


// КД1. Обчислення 2: a = min(a, ai)
#pragma omp critical(CS)
```

```
        {
            if (a_i < a) {
                a = a_i;
            }
        }

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

        // КД2. Копія ei = e
        #pragma omp critical(CS)
        {
            e_i = e;
        }

        // КД3. Копія ai = a
        #pragma omp critical(CS)
        {
            a_i = a;
        }

        // Обчислення 3: Ан = ai * (B * MVн) + ei * X * (MM * MCн)
        calculateStep3(MV, MM, MC, Z, B, X, a, e, A, tId);

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

        if (tId == 3) {
            #pragma omp critical
            {
                // Виведення фінального результату A в задачі T3
                std::cout << "A: [ ";
                printFinalVectorA(A, N);
                std::cout << "]" << std::endl;
            }
        }

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

    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout << "Time: " << duration.count() << " ms" << std::endl;

    clearMemory();

    return 0;
}

// Метод для заповнення скаляру
int fillScalar() {
    return 1;
}

// Метод для заповнення вектора
int* fillVector(int N) {
    int* res = new int[N];
    for (int i = 0; i < N; i++) {
        res[i] = 1;
    }
    return res;
}
```

```
// Метод для заповнення матриці
int** fillMatrix(int N) {
    int** res = new int* [N];
    for (int i = 0; i < N; i++) {
        res[i] = new int[N];
        for (int j = 0; j < N; j++) {
            res[i][j] = 1;
        }
    }
    return res;
}

// Метод для виведення фільного результату
void printFinalVectorA(int* vector, int N) {
    for (int i = 0; i < N; ++i) {
        std::cout << vector[i] << " ";
    }
}

// Метод для знаходження мінімуму вектора. Кострукція #pragma for
int findMinInSubVector(int* vector, int start, int end) {
    int min = vector[start];
    for (int i = start; i < end; i++) {
        if (min > vector[i]) {
            min = vector[i];
        }
    }
    return min;
}

// Метод для отримання підматриці
int** getSubMatrix(int** matrix, int partOfMtrx) {
    int** res = new int* [N];

    for (int i = 0; i < N; i++) {
        res[i] = new int[H];
    }

    int startPos = (partOfMtrx - 1) * H;
    int endPos = startPos + H;

    for (int j = startPos; j < endPos; ++j) {
        for (int i = 0; i < N; ++i) {
            res[i][j - startPos] = matrix[i][j];
        }
    }

    return res;
}

// Метод для множення матриць
int** multiplyMatrix(int** first_matrix, int** second_matrix) {
    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] += first_matrix[i][k] * second_matrix[k][j];
            }
        }
    }
```

```
    return res;
}

// Метод для множення вектора на матрицю
int* multiplyVectorAndMatrix(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;
}

// Метод для множення скаляру на вектор
int* multiplyScalarAndVector(int scalar, int* vector) {
    int* res = new int[N];

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

    return res;
}

// Метод для додавання векторів
int* addVectors(int* first_vector, int* second_vector) {
    int* res = new int[H];

    for (int i = 0; i < H; i++) {
        res[i] = first_vector[i] + second_vector[i];
    }

    return res;
}

// Метод для об'єднання підвекторів в один вектор
void combineVector(const int* vector, int* result, int partOfVctr) {
    int startPos = (partOfVctr - 1) * H;

    for (int i = 0; i < H; i++) {
        result[startPos + i] = vector[i];
    }
}

// Метод для обчислення кроку 3
void calculateStep3(int** MV, int** MM, int** MC, int* Z, int* B, int* X, int a, int e, int* A, int tId) {
    int** MV_H = getSubMatrix(MV, tId);
    int** MC_H = getSubMatrix(MC, tId);

    int* Ah = addVectors(
        multiplyScalarAndVector(
            a,
            multiplyVectorAndMatrix(B, MV_H)
        ),
        multiplyScalarAndVector(
            e,
            multiplyVectorAndMatrix(
                X,
                multiplyMatrix(MM, MC_H)
            )
```

```
        )
    );
    combineVector(Ah, A, tId);
    for (int i = 0; i < N; i++) {
        delete[] MV_H[i];
        delete[] MC_H[i];
    }
    delete[] MV_H;
    delete[] MC_H;
}

// Метод для очищення пам'яті
void clearMemory() {
    for (int i = 0; i < N; i++) {
        delete[] MV[i];
        delete[] MC[i];
        delete[] MM[i];
    }
    delete[] MV;
    delete[] MC;
    delete[] MM;
    delete[] A;
    delete[] B;
    delete[] X;
    delete[] Z;
}
```

Результат виконання програми для N = 16.

```
Thread_1 is started
Thread_3 is started
Thread_2 is started
Thread_4 is started
Thread_1 is finished
Thread_2 is finished
Thread_4 is finished
A: [ 272 272 272 272 272 272 272 272 272 272 272 272 272 272 272 272 ]
Thread_3 is finished
Time: 9 ms
```

Тестування:

Опис комп'ютера:

AMD Ryzen 7 4700U with Radeon Graphics

Базовая скорость:	2,00 ГГц
Сокетов:	1
Ядра:	8
Логических процессоров:	8

Щоб отримати більш точні результати, програма була запущеня кілька разів, і була обчислена середня швидкість програми для N = 1500.

Використовується pragma for

1 ядро:

1. 14361 ms

2. 14462 ms

3. 13596 ms

4. 14196 ms

5. 14211 ms

Середнє значення : 14165 ms

4 ядра:

1. 5905 ms

2. 5856 ms

3. 5685 ms

4. 5756 ms

5. 5722 ms

Середнє значення : 5785 ms

Коефіцієнт прискорення дорівнює:

$$K_{\text{у}} = \frac{T_1}{T_4} = \frac{14165}{5785} = 2{,}45$$

Не використовується pragma for

1 ядро:

1. 14816 ms

2. 14607 ms

3. 14803 ms

4. 14263 ms

5. 14570 ms

Середнє значення : 14611 ms

4 ядра:

1. 6073 ms

2. 5964 ms

3. 5895 ms

4. 6036 ms

5. 6106 ms

Середнє значення : 6015 ms

Коефіцієнт прискорення дорівнює:

$$K_{\text{у}} = \frac{T_1}{T_4} = \frac{14611}{6015} = 2{,}43$$

Висновок

1) У процесі розробки було створено паралельний математичний алгоритм, який включає в себе паралельне множення вектора на матрицю, множення двох матриць, множення скаляра на вектор, додавання векторів, а також знаходеження мінімуму вектора. Також, було визначено спільні ресурси (СР): скаляри a, e та вектори B, X, матриця MM.

2) Були розроблені алгоритми для потоків, в яких кожен потік виконує паралельне обчислення своєї частини задачі. Також було визначено точки синхронізації та критичні ділянки КД1-КД3.

3) Була розроблена схема взаємодії задач, в якій було визначено та відмічено засоби організації взаємодії потоків:

   Бар'єри - синхронізація взаємодії потоків.

   Критичні секції - захист спільних ресурсів

4) Лабораторну роботу виконано за допомогою мови програмування C++ та бібліотеки OpenMP для роботи з потоками. Для досягнення цього були використані наступні засоби та методи:
   - ***#pragma omp parallel*** – використовується для створення паралельних областей в коді, де відбувається виконання декількох потоків. Ключові слова ***private*** та ***shared*** використовуються для визначення, які змінні будуть приватними для кожного потоку та які змінні будуть спільними для всіх потоків.
   - ***omp_set_num_threads(P)*** – для встановлення кількості потоків, які будуть використовуватись у паралельному виконанні.
   - ***omp_get_thread_num()*** – для повернення номеру поточного потоку.
   - ***#pragma omp parallel for*** – використовується для автоматичного розподілу ітерацій циклу між потоками, дозволяючи ефективно використовувати паралельні потоки для обробки даних.
   - ***#pragma omp barrier*** – створює бар'єр, який забезпечує, що всі потоки досягли даної точки в коді перед тим, як будуть продовжені виконання далі.
   - ***#pragma omp critical*** – використовується для захисту критичних секцій коду, де доступ до спільних ресурсів може викликати проблеми конкурентної взаємодії між потоками.

5) В результаті проведеного тестування було підтверджено ефективність багатопотокової програми. При значенні N = 1500 отримані значення коефіцієнта прискорення складають 2,45 для програми з pragma for та 2,43 для програми без нього. Середня швидкість виконання програми з pragma for та без нього на чотирьох ядрах становить відповідно 14165 мс та 14611 мс, що свідчить про те, що обидві програми працюють майже з однаковою швидкістю.