

Отчет по лабораторной работе №3

Выполнил Герасимов АД, ИУСбд-01-20, 1132210569

Example 3.1

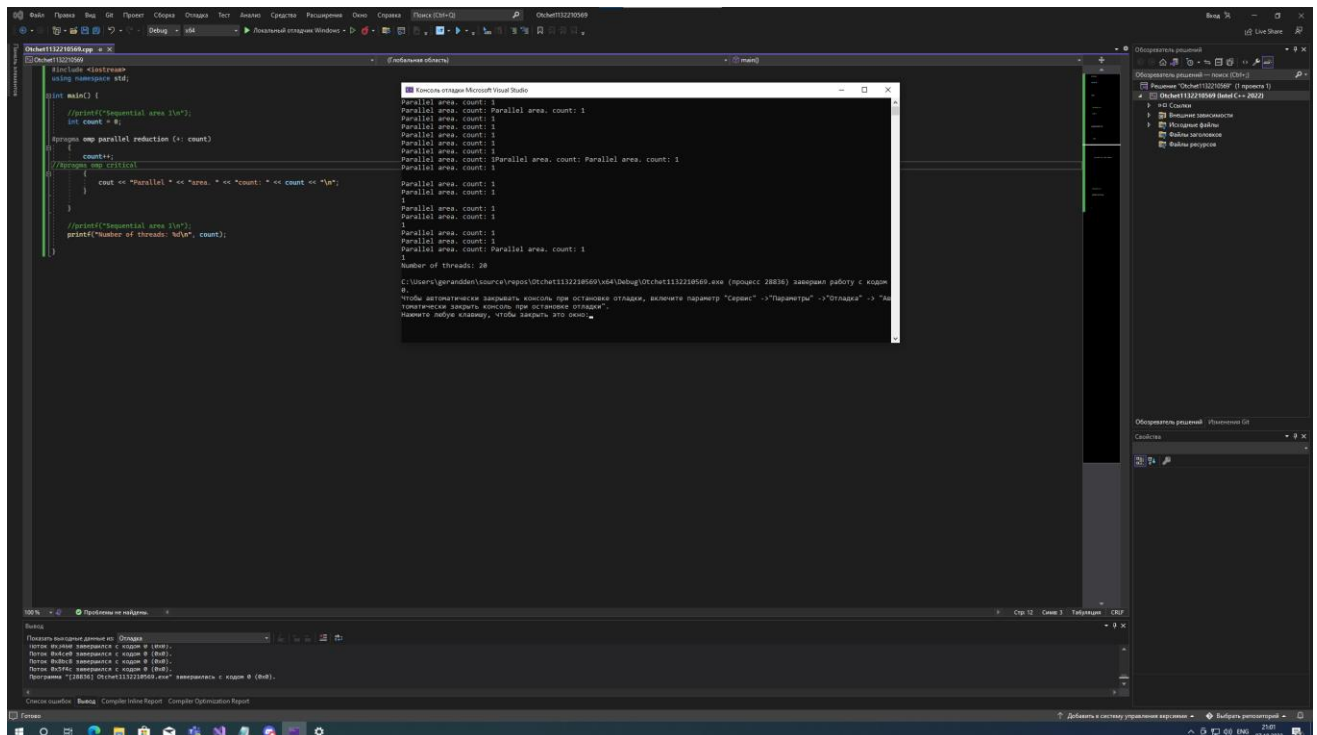
```
#include<iostream>
using namespace std;

int main(){

    printf("Sequential area 1\n");
    int count = 0;

    #pragma omp parallel reduction (+: count)
    {
        count++;
        cout << "Parallel " << "area. " << "count: " << count << "\n";
    }

    printf("Sequential area 1\n");
    printf("Number of threads: %d\n", count);
}
```



Упражнения.

1. Обратите внимание на ошибки (непоследовательность) при выводе сообщения внутри потоков.

Чем это вызвано? Почему этого не происходит при выводе результата работы программы? Как можно решить эту проблему?

(Подсказка: каким ресурсом хотят воспользоваться потоки и как организовать эксклюзивный доступ к нему.)

Как мы знаем, для решения этой проблемы можно изменить процесс вывода или защитить работу потока с этим ресурсом.

Для защиты (настройки приватного доступа) операции вывода в OpenMP можно выделить этот участок кода как критический, т.е.

работающий только с одним потоком. Для этого необходимо использовать следующую команду:

```
#pragma omp critical
{
    ...
}
```

Устраните ошибки (непоследовательность) при выводе сообщения внутри потоков.

*/

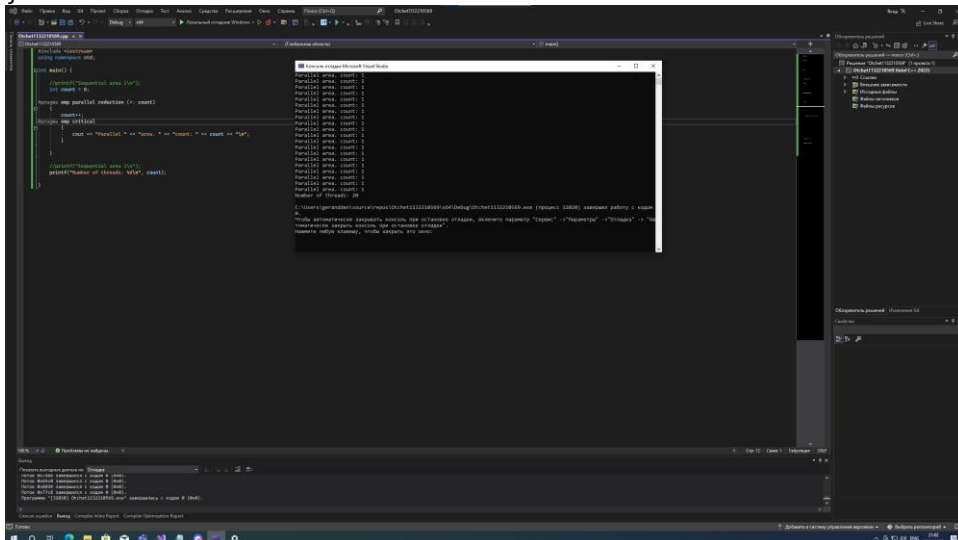
```
#include <iostream>
using namespace std;

int main(){

    //printf("Sequential area 1\n");
    int count = 0;

    #pragma omp parallel reduction (+: count)
    {
        count++;
        #pragma omp critical
        {
            cout << "Parallel " << "area. " << "count: " << count << "\n";
        }
    }

    //printf("Sequential area 1\n");
    printf("Number of threads: %d\n", count);
}
```



Example 3.2

Упражнения.

1. Напишите программу производящую параллельное суммирование элементов массива.

```
#include <iostream>
#include <ctime>

using namespace std;

int getSumArr(int* arr, int length) {
    int result = 0;

#pragma omp for
    for (int i = 0; i < length; ++i) {
#pragma omp atomic
        result += arr[i];
    }

    return result;
}

void print_arr(int* arr, int length) {
    cout << "{ ";

    for (int i = 0; i < length; i++) {
        if (i != length - 1) {
            cout << arr[i] << ", ";
        }
        else {
            cout << arr[i];
        }
    }

    cout << " }\n";
}

int main() {
    srand(time(NULL));
    cout << "\nLet's generate the array and get the amount of its elements!
(paralelly)\n";

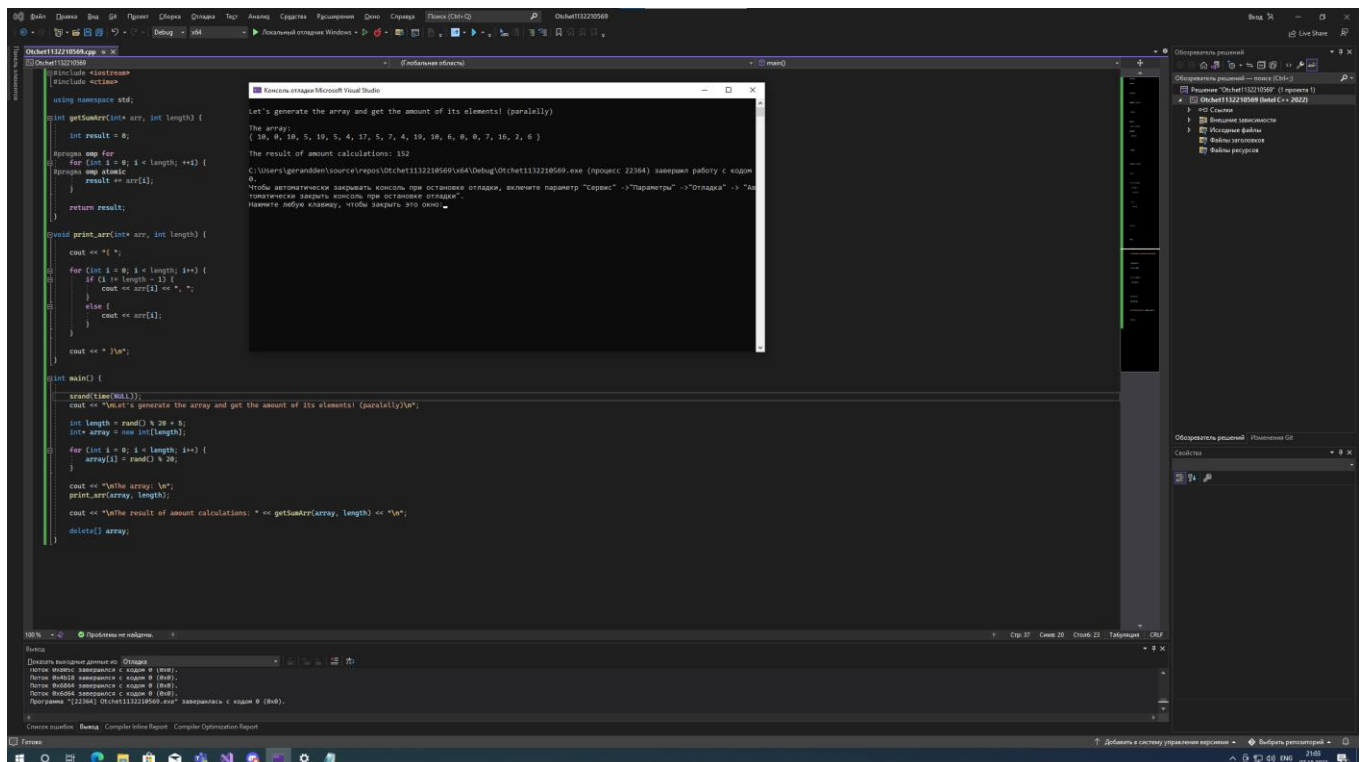
    int length = rand() % 20 + 5;
    int* array = new int[length];

    for (int i = 0; i < length; i++) {
        array[i] = rand() % 20;
    }

    cout << "\nThe array: \n";
    print_arr(array, length);

    cout << "\nThe result of amount calculations: " << getSumArr(array, length) << "\n";

    delete[] array;
}
```



Example 3.3

Упражнения.

1.1. Как видно, количество создаваемых потоков совпадает с числом логических ядер на устройстве.

Однако, это не всегда рационально, можно явно указать необходимое количество потоков, дополнив команду

```
#pragma omp parallel
```

соответствующей директивой, например, создав именно 3 потока

```
#pragma omp parallel num_threads(3)
```

Дополните этим выражением программу и посмотрите на результат.

```
#include <iostream>
```

```
#include "omp.h"
```

```
using namespace std;
```

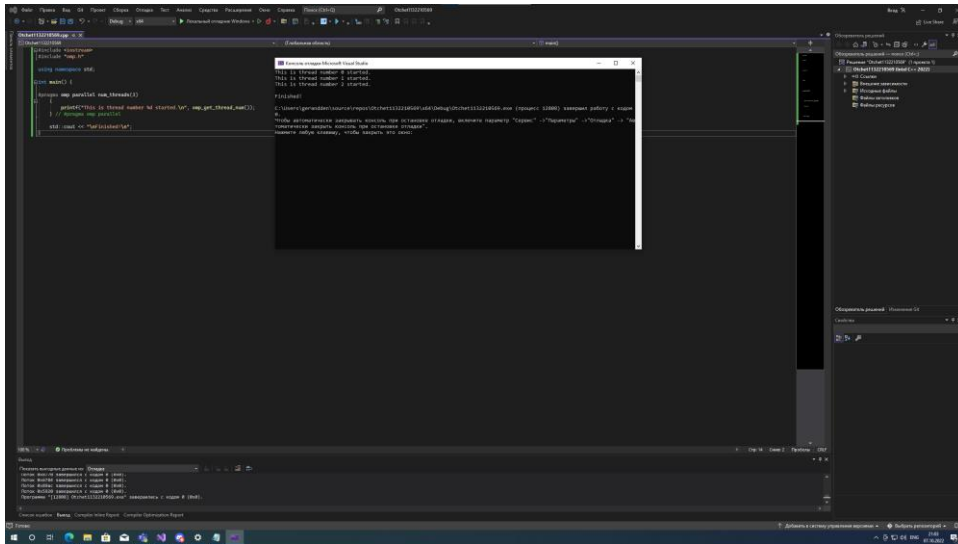
```
int main() {
```

```
    #pragma omp parallel num_threads(3)
    {
```

```
        printf("This is thread number %d started.\n", omp_get_thread_num());
    } // #pragma omp parallel
```

```
    std::cout << "\nFinished!\n";
```

```
}
```



1.2. Кроме того, часто бывает необходимым выполнять параллельно разные задачи. Для этого можно разделить параллельный код на разные задания для потоков, это делается с помощью директивы:

```
#pragma omp:llvm task
{
...
}
```

Поместите в блок `#pragma omp parallel num_threads(2) {...}` выражение:

```
    // task 1
    #pragma omp:llvm task
    {
        printf("This is thread number %d started.\n",
omp_get_thread_num());
    } // #task 1 end
```

и посмотрите на результат.

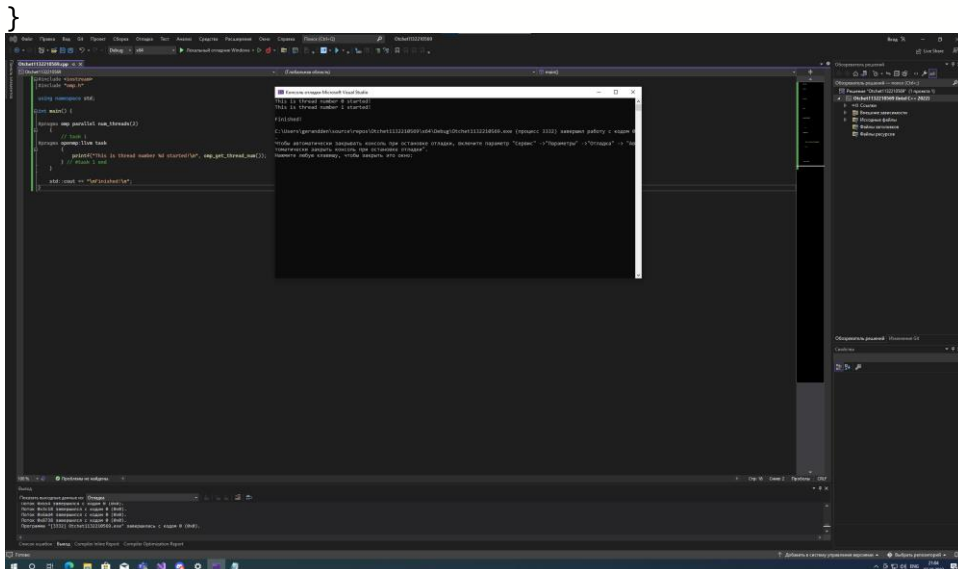
```
#include <iostream>
#include "omp.h"
```

```
using namespace std;
```

```
int main() {

    #pragma omp parallel num_threads(2)
    {
        // task 1
        #pragma omp:llvm task
        {
            printf("This is thread number %d started!\n", omp_get_thread_num());
        } // #task 1 end
    }

    std::cout << "\nFinished!\n";
```



1.3. Если необходимо раздать уникальные задачи потокам, необходимо использовать директиву

```
#pragma omp single{
...
}
```

Измените блок `#pragma omp parallel num_threads(2) {...}`

поместив туда выражение:

```
#pragma omp single
{
    // task 1
    #pragma omp:llvm task
    {
        printf("This is thread number %d started.\n",
omp_get_thread_num());
    } // #task 1 end
}
#pragma omp single
{
    // task 2
    #pragma omp:llvm task
    {
        printf("This is a different thread number %d started.\n",
omp_get_thread_num());
    } // #task 2 end
}
```

```
#include <iostream>
```

```
#include "omp.h"
```

```
using namespace std;
```

```
int main() {
```

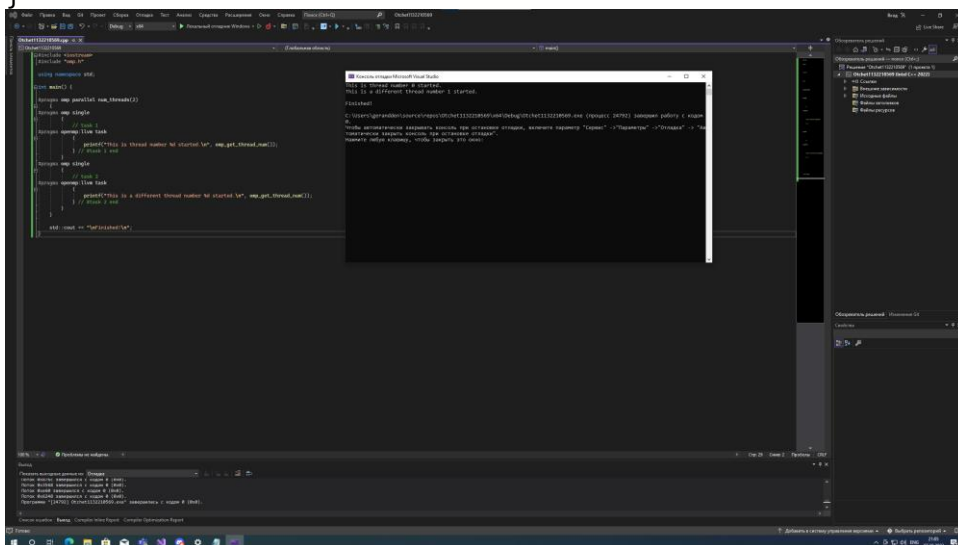
```
    #pragma omp parallel num_threads(2)
```

```

{
    #pragma omp single
    {
        // task 1
        #pragma openmp:llvm task
        {
            printf("This is thread number %d started.\n",
omp_get_thread_num());
        } // #task 1 end
    }
    #pragma omp single
    {
        // task 2
        #pragma openmp:llvm task
        {
            printf("This is a different thread number %d started.\n",
omp_get_thread_num());
        } // #task 2 end
    }
}

std::cout << "\nFinished!\n";
}

```



2. Создайте 5 различных параллельных задач выводящих на экран числа от 1 до 5. Проанализируйте полученную последовательность чисел.

```

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

using namespace std;

int main() {

#pragma omp parallel num_threads(5)
{

```

```
#pragma omp single
{
    // task 1
#pragma omp:llvm task
    {
        cout << 1;
    } // #task 1 end
}
```

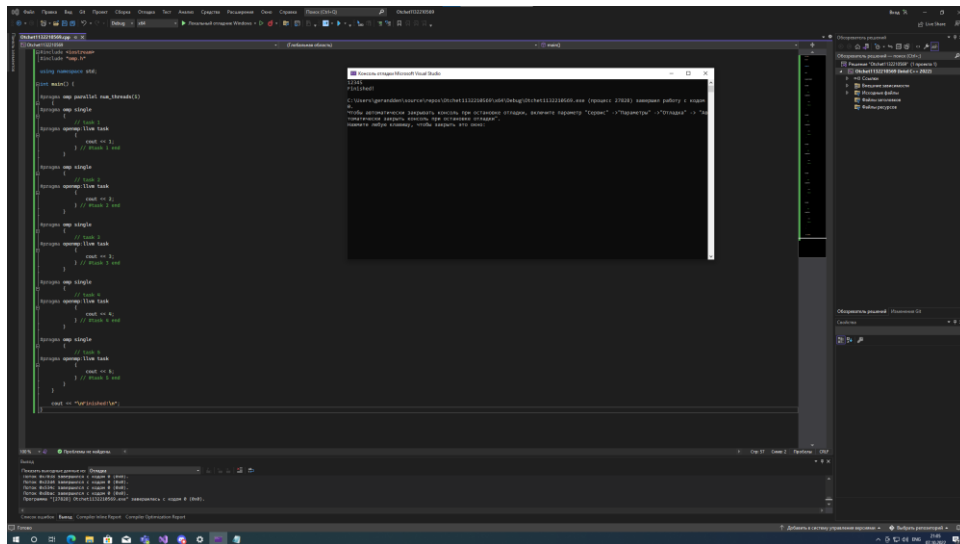
```
#pragma omp single
{
    // task 2
#pragma omp:llvm task
    {
        cout << 2;
    } // #task 2 end
}
```

```
#pragma omp single
{
    // task 3
#pragma omp:llvm task
    {
        cout << 3;
    } // #task 3 end
}
```

```
#pragma omp single
{
    // task 4
#pragma omp:llvm task
    {
        cout << 4;
    } // #task 4 end
}
```

```
#pragma omp single
{
    // task 5
#pragma omp:llvm task
    {
        cout << 5;
    } // #task 5 end
}
```

```
    cout << "\nFinished!\n";
}
```

3.1. Напишите программу состоящую из трёх потоков работающих с вектором данных из чисел типа `int`:

- первый поток последовательно записывает в вектор от 0 до `n`,
 - второй последовательно удаляет значения с конца вектора
 - третий выводит состояние вектора после хотябы одного изменения его состояния
- Работа программы завершается когда вектор оказывается пустым.

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

bool flag1 = 1, flag2 = 1, flag3 = 1, flag4 = 1;

template <typename T>
class container {

    mutex mtx;
    vector<T> vector;

public:

    void add(T element) {
        mtx.lock();
        vector.push_back(element);
        mtx.unlock();
    }

    void print() {
        mtx.lock();
        cout << "Vector: | ";
        for (auto e : vector) {
            cout << e << " ";
        }
        cout << "|" << endl;
        mtx.unlock();
    }

    void pop() {
        mtx.lock();
        vector.pop_back();
    }
};
```

```

        mtx.unlock();
    }

    bool chek_vec() {
        if (!vector.empty())
            return true;
        else
            return false;
    }
};

void write(container<int>& c, int n) {

    for (int i = 0; i <= n; i++) {
        c.add(i);
        flag1 = 1;
        flag2 = 0;
        do {
            this_thread::yield();
        } while (flag1);
    }

    flag3 = 0;

}

void clean(container<int>& c, int n) {
    do {
        this_thread::yield();
    } while (flag3);

    while (c.chek_vec()) {
        do {
            this_thread::yield();
        } while (flag1);
        c.pop();
        flag2 = 0;
        flag1 = 1;
    }

    flag4 = 0;
    flag2 = 0;

}

void read(container<int>& c, int n) {
    while (true) {
        do {
            this_thread::yield();
        } while (flag2);
        if (flag4 == 0) {
            break;
        }
        c.print();
        flag1 = 0;
        flag2 = 1;
    }
}

int main() {

    srand(time(NULL));

    int length;

```

```

    cout << "The length of your vector: ";
    length = 8;
    cout << length;
    cout << "\n\n";

    container<int> vector;

    do {} while (!flag1 && !flag2 && !flag4 && flag3 != 0);

#pragma omp parallel sections num_threads(3)
{
#pragma omp section // Reading
{
    //cout << "\nR - start\n";
    read(ref(vector), length);
    //cout << "\nR - end\n";

}

#pragma omp section // Writing
{
    //cout << "\nW - start\n";
    write(ref(vector), length);
    //cout << "\nW - end\n";

}

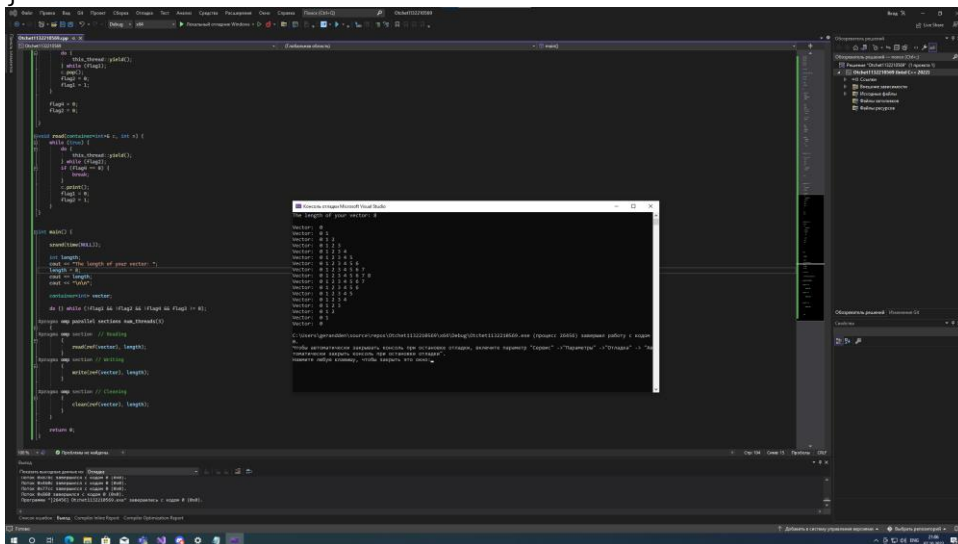
#pragma omp section // Cleaning
{
    //cout << "\nC - start\n";
    clean(ref(vector), length);
    //cout << "\nC - end\n";

}

}

return 0;
}

```



3.2. Увеличьте количество пишущих и стирающих потоков в предыдущей программе.

```

#include <iostream>
#include <cstdlib>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

```

```
bool flag1 = 1, flag2 = 1, flag3 = 1, flag4 = 1, flag5 = 0;
int i_write = 0;
```

```
template <typename T>
class container {
```

```
    mutex mtx;
    vector<T> vector;
```

```
public:
```

```
    void add(T element) {
        mtx.lock();
        vector.push_back(element);
        mtx.unlock();
    }
```

```
    void print() {
        mtx.lock();
        cout << "Vector: | ";
        for (auto e : vector) {
            cout << e << " ";
        }
        cout << "|" << endl;
        mtx.unlock();
    }
```

```
    void pop() {
        mtx.lock();
        vector.pop_back();
        mtx.unlock();
    }
```

```
    bool chek_vec() {
        if (!vector.empty())
            return true;
        else
            return false;
    }
```

```
};
```

```
void write(container<int>& c, int n) {
```

```
    for (; i_write <= n;) {
        c.add(i_write);
        flag1 = 1;
        flag2 = 0;
        do {
            this_thread::yield();
        } while (flag1);
```

```
#pragma omp atomic;
```

```
    i_write++;
}
```

```
    flag3 = 0;
```

```
}
```

```
void clean(container<int>& c, int n) {
```

```
    do {
        this_thread::yield();
    } while (flag3);
```

```
#pragma omp critical
```

```

{
    while (c.chek_vec()) {

        if (flag5) {
            flag2 = 0;
            flag4 = 0;
            break;
        }

        do {
            this_thread::yield();
        } while (flag1);

        c.pop();

        if (!c.chek_vec()) {
            flag5 = 1;
        }

        if (flag5) {
            flag2 = 0;
            flag4 = 0;
            break;
        }

        flag2 = 0;
        flag1 = 1;
    }
}

flag4 = 0;
}

void read(container<int>& c, int n) {
    while (true) {
        do {
            this_thread::yield();
        } while (flag2);
        if (flag4 == 0) {
            break;
        }
        c.print();
        flag1 = 0;
        flag2 = 1;
    }
}

int main() {

    srand(time(NULL));

    int length;
    cout << "The length of your vector: ";
    length = rand() % 15 + 5;
    cout << length;
    cout << "\n\n";

    container<int> vector;

    do {} while (!flag1 && !flag2 && !flag4 && flag3 != 0 && !i_write && !flag5);

#pragma omp parallel sections num_threads(3)

```

```

{
#pragma omp section // Reading
{
    //cout << "\nR - start\n";
    read(ref(vector), length);
    //cout << "\nR - end\n";
}

#pragma omp section // Writing
{
    //cout << "\nW1 - start\n";
    write(ref(vector), length);
    //cout << "\nW1 - end\n";
}

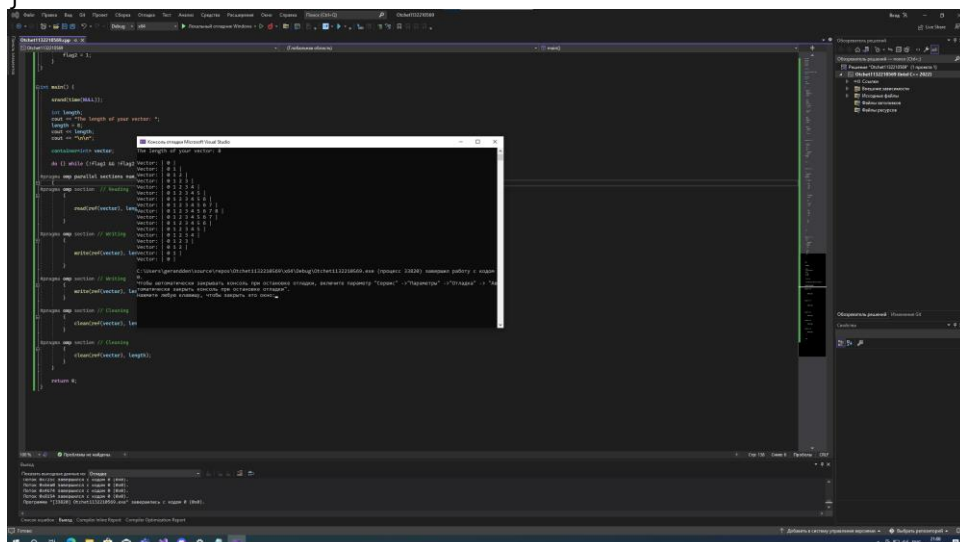
#pragma omp section // Writing
{
    //cout << "\nW2 - start\n";
    write(ref(vector), length);
    //cout << "\nW2 - end\n";
}

#pragma omp section // Cleaning
{
    //cout << "\nC2 - start\n";
    clean(ref(vector), length);
    //cout << "\nC2 - end\n";
}

#pragma omp section // Cleaning
{
    //cout << "\nC1 - start\n";
    clean(ref(vector), length);
    //cout << "\nC1 - end\n";
}
}

return 0;
}

```



Домашнее задание(базовое)

1. Напишите программу производящую параллельное суммирование двух матриц. Результат записывается в матрице продукт и выводится на экран.

```
#include <iostream>

using namespace std;

void printMatrix(int** matrix, int count_rows, int count_columns) {
    for (int i = 0; i < count_rows; i++) {
        for (int j = 0; j < count_columns; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

void SummTheRows(int* row1, int* row2, int count_elements, int* row_result) {
#pragma omp for
    for (int i = 0; i < count_elements; i++) {
        row_result[i] = row1[i] + row2[i];
    }
}

int main() {
    srand(time(NULL));

    int n, m, first_count_rows, first_count_columns, second_count_rows,
    second_count_columns;
    int** first_matrix;
    int** second_matrix;
    int** result_matrix;

    cout << "Enter count of rows of matrices: ";
    cin >> n;
    cout << "Enter count of columns of matrices: ";
    cin >> m;

    cout << endl;

    first_count_rows = n;
    first_count_columns = m;
    first_matrix = new int* [first_count_rows];

    for (int i = 0; i < first_count_rows; ++i) {
        first_matrix[i] = new int[first_count_columns];
    }

    for (int i = 0; i < first_count_rows; i++) {
        for (int j = 0; j < first_count_columns; j++) {
            first_matrix[i][j] = rand() % 1000;
        }
    }

    cout << endl;
    cout << "First matrix: " << endl;
    printMatrix(first_matrix, first_count_rows, first_count_columns);

    second_count_rows = n;
    second_count_columns = m;
    second_matrix = new int* [second_count_rows];

    for (int i = 0; i < second_count_rows; ++i) {
        second_matrix[i] = new int[second_count_columns];
    }
}
```

```

    }

    for (int i = 0; i < second_count_rows; i++) {
        for (int j = 0; j < second_count_columns; j++) {
            second_matrix[i][j] = rand() % 1000;
        }
    }

    cout << endl;
    cout << "Second matrix: " << endl;
    printMatrix(second_matrix, second_count_rows, second_count_columns);

    int count_rows = n;
    int count_columns = m;

    result_matrix = new int* [count_rows];
    for (int i = 0; i < count_rows; ++i) {
        result_matrix[i] = new int[count_columns];
    }

#pragma omp for
    for (int i = 0; i < count_rows; i++) {
        SummTheRows(first_matrix[i], second_matrix[i], count_columns,
result_matrix[i]);
    }

    cout << endl;
    cout << "Getting amount... Result matrix: " << endl;
    printMatrix(result_matrix, count_rows, count_columns);

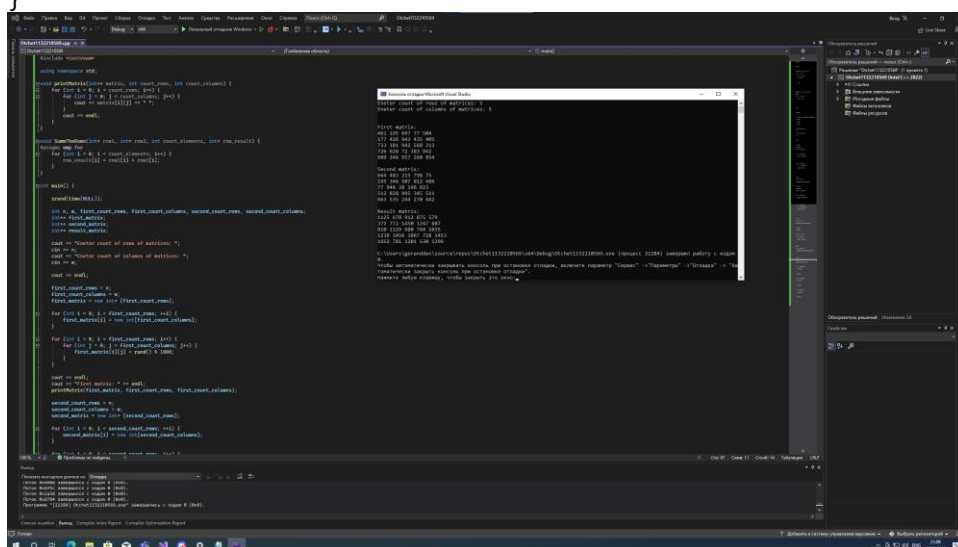
    for (int i = 0; i < second_count_rows; ++i) {
        delete[] second_matrix[i];
    }
    delete[] second_matrix;

    for (int i = 0; i < first_count_rows; ++i) {
        delete[] first_matrix[i];
    }
    delete[] first_matrix;

    for (int i = 0; i < count_rows; ++i) {
        delete[] result_matrix[i];
    }
    delete[] result_matrix;

    return 0;
}

```



The screenshot shows a C++ IDE with the following output in the console:

```

First matrix:
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10

Second matrix:
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10

Result matrix:
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10
10 10

```


2. Напишите программу производящую параллельное умножение двух матриц. Результат записывается в матрице продукт и выводится на экран.

```
#include <iostream>
#include <cmath>

using namespace std;

void printMatrix(int** matrix, int count_rows, int count_columns) {
    for (int i = 0; i < count_rows; i++) {
        for (int j = 0; j < count_columns; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

int GetComposition(int* m1, int* m2, int c) {
    int num = 0;

#pragma omp for
    for (int i = 0; i < c; i++) {
        num += m1[i] * m2[i];
    }

    return num;
}

int main() {
    srand(time(NULL));

    int n, m, first_count_rows, first_count_columns, second_count_rows,
    second_count_columns;
    int** first_matrix;
    int** second_matrix;
    int** result_matrix;

    cout << "Enter count of rows of matrices: ";
    cin >> n;
    cout << "Enter count of columns of matrices: ";
    cin >> m;

    cout << endl;

    first_count_rows = n;
    first_count_columns = m;
    first_matrix = new int* [first_count_rows];

    for (int i = 0; i < first_count_rows; ++i) {
        first_matrix[i] = new int[first_count_columns];
    }

    for (int i = 0; i < first_count_rows; i++) {
        for (int j = 0; j < first_count_columns; j++) {
            first_matrix[i][j] = rand() % 50;
        }
    }

    cout << endl;
    cout << "First matrix: " << endl;
    printMatrix(first_matrix, first_count_rows, first_count_columns);
```

```

second_count_rows = m;
second_count_columns = n;
second_matrix = new int* [second_count_rows];

for (int i = 0; i < second_count_rows; ++i) {
    second_matrix[i] = new int[second_count_columns];
}

for (int i = 0; i < second_count_rows; i++) {
    for (int j = 0; j < second_count_columns; j++) {
        second_matrix[i][j] = rand() % 50;
    }
}

cout << endl;
cout << "Second matrix: " << endl;
printMatrix(second_matrix, second_count_rows, second_count_columns);

int** T_matrix = new int* [second_count_columns];
for (int i = 0; i < second_count_columns; ++i) {
    T_matrix[i] = new int[second_count_rows];
}

for (int i = 0; i < second_count_rows; i++) {
    for (int j = 0; j < second_count_columns; j++) {
        T_matrix[j][i] = second_matrix[i][j];
    }
}

int count_rows = first_count_rows;
int count_columns = second_count_columns;

result_matrix = new int* [count_rows];
for (int i = 0; i < count_rows; ++i) {
    result_matrix[i] = new int[count_columns];
}

#pragma omp for
{
    for (int i = 0; i < count_rows; i++) {
        for (int z = 0, j = 0; j < count_columns; j++, z++) {
            result_matrix[i][j] = GetComposition(first_matrix[i],
T_matrix[z], count_columns);
        }
    }
}

cout << endl;
cout << "Getting composition... Result matrix: " << endl;
printMatrix(result_matrix, count_rows, count_columns);

for (int i = 0; i < second_count_rows; ++i) {
    delete[] second_matrix[i];
}
delete[] second_matrix;

for (int i = 0; i < first_count_rows; ++i) {
    delete[] first_matrix[i];
}
delete[] first_matrix;

for (int i = 0; i < second_count_columns; ++i) {
    delete[] T_matrix[i];
}

```

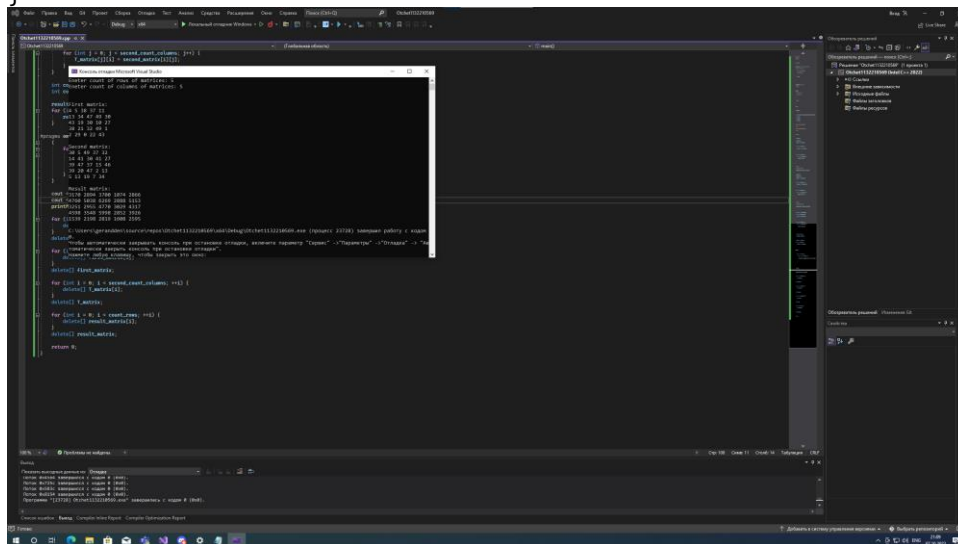
```

delete[] T_matrix;

for (int i = 0; i < count_rows; ++i) {
    delete[] result_matrix[i];
}
delete[] result_matrix;

return 0;
}

```



3. Ранее мы рассматривали (лекция 3) алгоритм вычисления чисел фибоначи на основе матричного произведения и быстрого возведения в степень.

```
def Fibonacci(n):
```

```

    p, q, r = 1, 1, 0
    N = n//2; par = n%2
    while N > 0:

```

	# Thread_1	Thread_2	Thread_3	Thread_4	
	pp = p*p;	qq = q*q;	rr = r*r;	pr = p + r;	# Такт
1					
барьер					
	p1 = pp + qq;	r1 = qq + rr;	q1 = q*pr;		# Такт
2					
барьер					
	p2 = p1 + q1;				# Такт
3					
барьер					
	if N % 2 == 1: p, q, r = p2, p1, q1				
	else: p, q, r = p1, q1, r1				
	N = N // 2				
	if par==0: return r				
	else: return q				

Напишите программу использующую параллельные потоки для реализации этого алгоритма и оцените время работы.

```
#include <iostream>
#include <cstdlib>
#include <thread>

using namespace std;
int flag1 = 1;
int flag2 = 1;
int flag3 = 1;
int flag4 = 1;

void func1(int& p, int& pp, int& p1, int& qq, int& p2, int& q1) {
    pp = p * p;

    do {
        this_thread::yield();
    } while (flag1);

    p1 = pp + qq;

    do {
        this_thread::yield();
    } while (flag3);

    p2 = p1 + q1;
}

void func2(int& qq, int& rr, int& r1, int& q) {
    qq = q * q;
    flag1 = 0;

    do {
        this_thread::yield();
    } while (flag2);

    r1 = qq + rr;
}

void func3(int& q1, int& rr, int& r, int& q, int& pr) {
    rr = r * r;
    flag2 = 0;

    do {
        this_thread::yield();
    } while (flag4);

    q1 = q * pr;
    flag3 = 0;
}

void func4(int& pr, int& p, int& r) {
    pr = p + r;
    flag4 = 0;
}

int fibonacci(int& n) {
    n -= 1;
    int p = 1, q = 1, r = 0, N = n / 2, par = n % 2;
    int pp, p1, p2, qq, r1, rr, q1, pr;
    int k = 0;

    while (N > 0) {
```

```

#pragma omp parallel sections num_threads(4)
{
#pragma omp section
{
    func1(ref(p), ref(pp), ref(p1), ref(qq), ref(p2), ref(q1));
    k++;
}

#pragma omp section
{
    func2(ref(qq), ref(rr), ref(r1), ref(q));
    k++;
}

#pragma omp section
{
    func3(ref(q1), ref(rr), ref(r), ref(q), ref(pr));
    k++;
}

#pragma omp section
{
    func4(ref(pr), ref(p), ref(r));
    k++;
}
}

if (k == 4) {
    flag1 = 1;
    flag2 = 1;
    flag3 = 1;
    flag4 = 1;
}

if (N % 2 == 1) {
    p = p2;
    q = p1;
    r = q1;
}

else {
    p = p1;
    q = q1;
    r = r1;
}

N = N / 2;
}

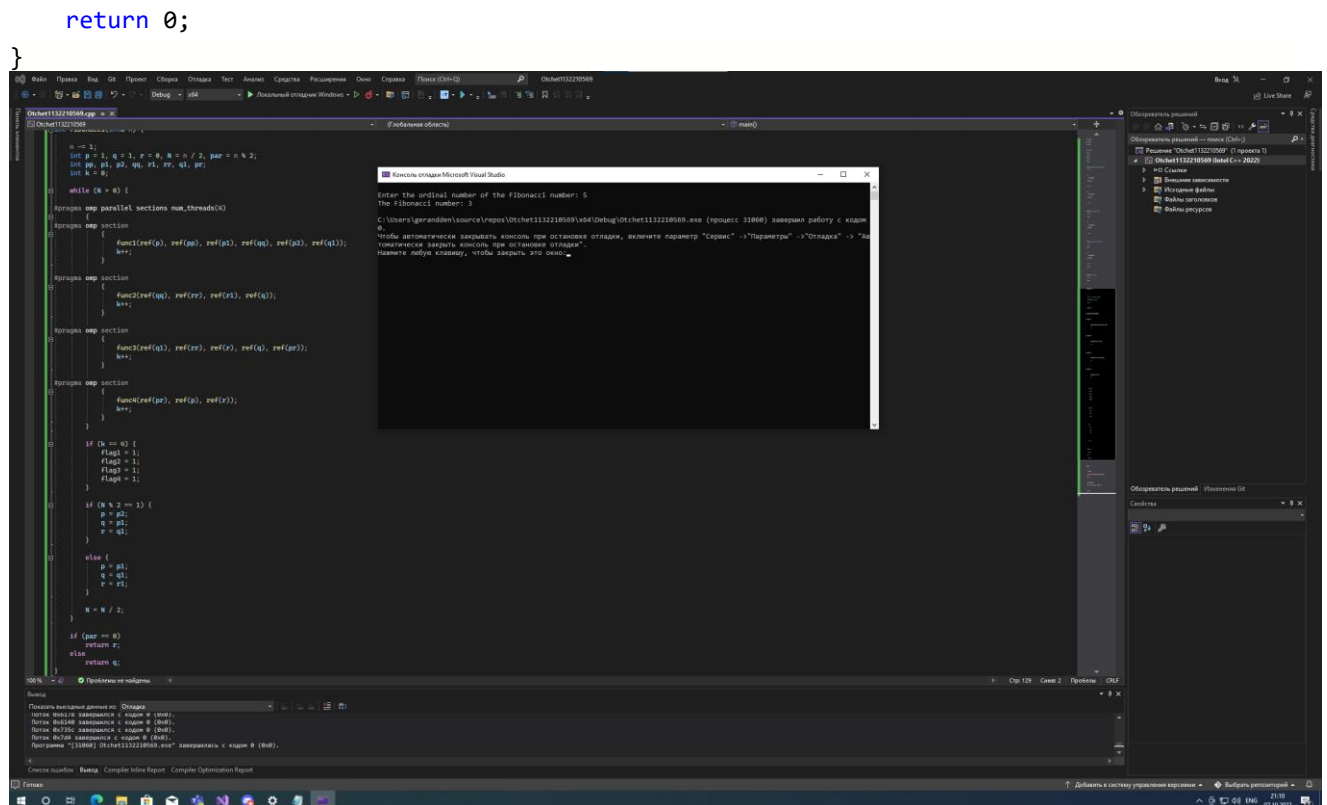
if (par == 0)
    return r;
else
    return q;
}

int main() {

    int n, result;
    cout << "\nEnter the ordinal number of the Fibonacci number: ";
    cin >> n;

    result = fibonacci(n);
    cout << "The Fibonacci number: " << result << "\n";
}

```



4. Ранее мы рассматривали алгоритм поиска в ширину на ориентированном не взвешенном графе. Состоящего из функций

```

def bfs(graph, s, out=0):
    parents = {v: None for v in graph}
    level = {v: None for v in graph}
    level[s] = 0
    queue = [s]
    while queue:
        v = queue.pop(0)
        for w in graph[v]:
            if level[w] is None:
                queue.append(w)
                parents[w] = v
                level[w] = level[v] + 1
        if out: print(level[w], level, queue)
    return level, parents

def PATH (end, parents):
    path = [end]
    parent = parents[end]
    while not parent is None:
        path.append(parent)
        parent = parents[parent]
    return path[::-1]

```

Рассмотрите возможность распаралеливания этого алгоритма и напишите соответствующую программу с помощью OpenMP.

```

*/

#include <iostream>

using namespace std;

void doSearch(int level[], const int n, int adj_v_w, int q[], int start, int& ends, int v) {
    if (level[adj_v_w] == -1) {
#pragma omp critical
    {
        q[ends++] = adj_v_w;
        level[adj_v_w] = level[v] + 1;
        cout << level[adj_v_w] << " ";

        for (int i = 0; i < n - 1; i++) {
            cout << level[i] << ", ";
        }

        cout << level[n - 1] << "]" << " ";

        for (int i = start; i < ends - 1; i++) {
            cout << q[i] << ", ";
        }

        cout << q[ends - 1] << "]" << endl;
    }
}

int main() {
    const int n = 6;
    int adj[n][3] = {
        {1, 2},
        {3, 4},
        {1, 4},
        {4},
        {1, 3, 5},
        {0, 2} };

    int level[n] = { -1, -1, -1, -1, -1, -1 };
    int s = 0;
    int q[n];
    int start = 0, ends = 0;
    int v;

    level[s] = 0;
    q[ends++] = s;

    while (start != ends) {
        v = q[start];
        start++;

#pragma omp for
        for (int w = 0; w < sizeof(adj[v]) / sizeof(adj[v][0]); w++) {
            doSearch(level, n, adj[v][w], q, start, ref(ends), v);
        }
    }
}

```

