


Задание к занятию 3

Теверовский Михаил, ИВТ-11М

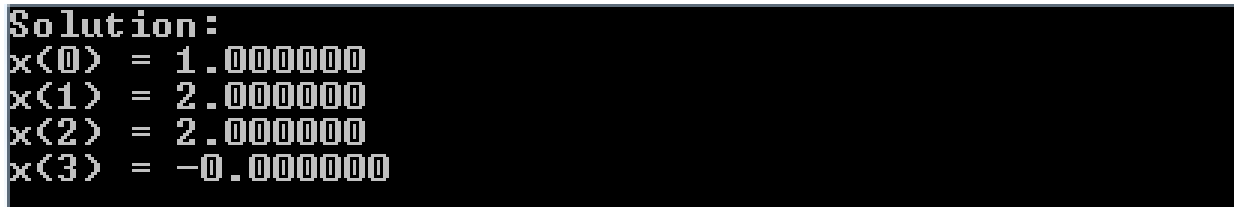
Используемые программы и характеристики ПК:

1. ПО: VisualStudio 2017 и Intel Parallel Studio 2019;
2. ОС: Windows 7 Professional
3. Процессор: Intel core i5-4200 CPU, 2.50 GHz, 2 ядра, 4 потока

Задание 1: В файле [task for lecture3.cpp](#)  приведен код, реализующий последовательную версию метода Гаусса для решения СЛАУ. Проанализируйте представленную программу.

Результат:

Убедимся в корректности работы:




```
Solution:
x(0) = 1.000000
x(1) = 2.000000
x(2) = 2.000000
x(3) = -0.000000
```

Рисунок 1. – Скриншот результата выполнения задания 1

Анализ полученных результатов:

Для проверки корректности выполнения программы сверим результаты с пакетом программ Matlab. Код в Matlab:



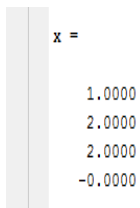
```
clc;
clear;

A = [2 5 4 1;
     1 3 2 1;
     2 10 9 7;
     3 8 9 2];

b = [20 11 40 37];

x = A^(-1)*b'
```

Результат выполнения кода:



```
x =
    1.0000
    2.0000
    2.0000
   -0.0000
```

Рисунок 2. – Скриншот результата выполнения кода в Matlab

Итак, мы получили такой же результат. Код работает корректно.

Задание 2: Запустите первоначальную версию программы и получите решение для тестовой матрицы *test_matrix*, убедитесь в правильности приведенного алгоритма. Добавьте строки кода для измерения времени (см. задание к занятию 2) выполнения прямого хода метода Гаусса в функцию *SerialGaussMethod()*. Заполните матрицу с количеством строк *MATRIX_SIZE* случайными значениями, используя функцию *InitMatrix()*. Найдите решение СЛАУ для этой матрицы (закомментируйте строки кода, где используется тестовая матрица *test_matrix*).

Результат:

1) Запустите первоначальную версию программы и получите решение для тестовой матрицы *test_matrix*

Результат представлен в «Задание 1».

2) Убедитесь в правильности приведенного алгоритма

Результат представлен в «Задание 1».

3) Добавьте строки кода для измерения времени (см. задание к занятию 2) выполнения прямого хода метода Гаусса в функцию *SerialGaussMethod()*:

Добавленные строки в коде ниже:

```
void SerialGaussMethod(double **matrix, const int rows, double* result)
{
    int k;
    double koef;
    high_resolution_clock::time_point t1, t2;
    t1 = high_resolution_clock::now();
    // прямой ход метода Гаусса
    for (k = 0; k < rows; ++k)
    {
        //
        for (int i = k + 1; i < rows; ++i)
        {
            koef = -matrix[i][k] / matrix[k][k];

            for (int j = k; j <= rows; ++j)
            {
                matrix[i][j] += koef * matrix[k][j];
            }
        }
    }
    t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
    printf("Time is: %lf sec\n\n", duration.count());
}
```

4) Заполните матрицу с количеством строк *MATRIX_SIZE* случайными значениями, используя функцию *InitMatrix()*

Код:

```

#include <stdio.h>
#include <ctime>
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#include <chrono>

using namespace std::chrono;

// количество строк в исходной квадратной матрице
const int MATRIX_SIZE = 1500;

/// Функция InitMatrix() заполняет переданную в качестве
/// параметра квадратную матрицу случайными значениями
/// matrix - исходная матрица СЛАУ
void InitMatrix(double** matrix)
{
    for (int i = 0; i < MATRIX_SIZE; ++i)
    {
        matrix[i] = new double[MATRIX_SIZE + 1];
    }

    for (int i = 0; i < MATRIX_SIZE; ++i)
    {
        for (int j = 0; j <= MATRIX_SIZE; ++j)
        {
            matrix[i][j] = rand() % 2500 + 1;
        }
    }
}

/// Функция SerialGaussMethod() решает СЛАУ методом Гаусса
/// matrix - исходная матрица коэффициентов уравнений, входящих в СЛАУ,
/// последний столбец матрицы - значения правых частей уравнений
/// rows - количество строк в исходной матрице
/// result - массив ответов СЛАУ
void SerialGaussMethod(double **matrix, const int rows, double* result)
{
    int k;
    double koef;
    high_resolution_clock::time_point t1, t2;
    t1 = high_resolution_clock::now();
    // прямой ход метода Гаусса
    for (k = 0; k < rows; ++k)
    {
        //
        for (int i = k + 1; i < rows; ++i)
        {
            koef = -matrix[i][k] / matrix[k][k];

            for (int j = k; j <= rows; ++j)
            {
                matrix[i][j] += koef * matrix[k][j];
            }
        }
    }
    t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
    printf("Time is:: %lf sec\n\n", duration.count());
    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

    for (k = rows - 2; k >= 0; --k)
    {
        result[k] = matrix[k][rows];
    }
}

```

```

        //
        for (int j = k + 1; j < rows; ++j)
        {
            result[k] -= matrix[k][j] * result[j];
        }

        result[k] /= matrix[k][k];
    }
}

int main()
{
    srand((unsigned)time(0));

    int i;

    // кол-во строк в матрице, приводимой в качестве примера
    //const int test_matrix_lines = 4;
    //Меняем на:
    const int test_matrix_lines = MATRIX_SIZE;

    double **test_matrix = new double*[test_matrix_lines];

    // цикл по строкам
    for (i = 0; i < test_matrix_lines; ++i)
    {
        // (test_matrix_lines + 1)- количество столбцов в тестовой матрице,
        // последний столбец матрицы отведен под правые части уравнений, входящих в
        СЛАУ
        test_matrix[i] = new double[test_matrix_lines + 1];
    }

    // массив решений СЛАУ
    double *result = new double[test_matrix_lines];

    // инициализация тестовой матрицы
    /*test_matrix[0][0] = 2; test_matrix[0][1] = 5; test_matrix[0][2] = 4;
    test_matrix[0][3] = 1; test_matrix[0][4] = 20;
    test_matrix[1][0] = 1; test_matrix[1][1] = 3; test_matrix[1][2] = 2;
    test_matrix[1][3] = 1; test_matrix[1][4] = 11;
    test_matrix[2][0] = 2; test_matrix[2][1] = 10; test_matrix[2][2] = 9;
    test_matrix[2][3] = 7; test_matrix[2][4] = 40;
    test_matrix[3][0] = 3; test_matrix[3][1] = 8; test_matrix[3][2] = 9;
    test_matrix[3][3] = 2; test_matrix[3][4] = 37; */
    //меняем на:
    InitMatrix(test_matrix);
    SerialGaussMethod(test_matrix, test_matrix_lines, result);

    for (i = 0; i < test_matrix_lines; ++i)
    {
        delete[] test_matrix[i];
    }

    printf("Solution:\n");

    for (i = 0; i < test_matrix_lines; ++i)
    {
        printf("x(%d) = %lf\n", i, result[i]);
    }

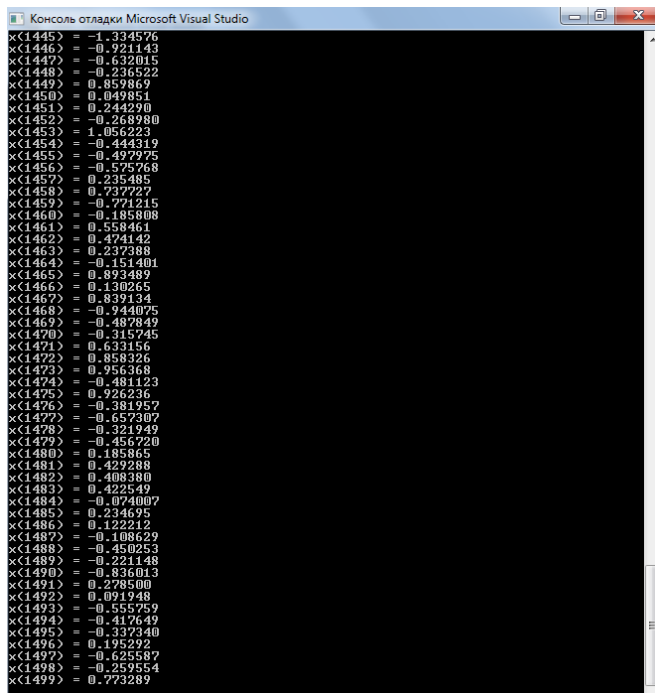
    delete[] result;

    return 0;
}

```

5) Найдите решение СЛАУ для этой матрицы (закомментируйте строки кода, где используется тестовая матрица *test_matrix*).

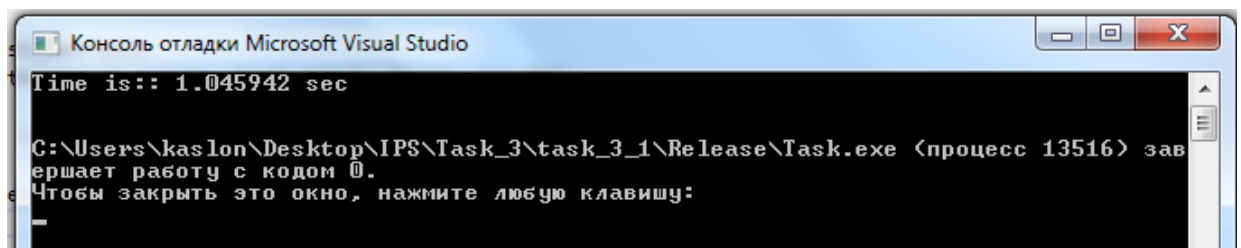
Фрагмент полученных значений:



```
Консоль отладки Microsoft Visual Studio
x(1445) = -1.334576
x(1446) = -0.921143
x(1447) = -0.632015
x(1448) = -0.236522
x(1449) = 0.859869
x(1450) = 0.049851
x(1451) = 0.244290
x(1452) = -0.268980
x(1453) = 1.056223
x(1454) = -0.443319
x(1455) = -0.497975
x(1456) = -0.575768
x(1457) = 0.235485
x(1458) = 0.737727
x(1459) = -0.771215
x(1460) = -0.185808
x(1461) = 0.558461
x(1462) = 0.474142
x(1463) = 0.237388
x(1464) = -0.151401
x(1465) = 0.893489
x(1466) = 0.130265
x(1467) = 0.839134
x(1468) = -0.944075
x(1469) = -0.487849
x(1470) = -0.315745
x(1471) = 0.633156
x(1472) = 0.858326
x(1473) = 0.956368
x(1474) = -0.481123
x(1475) = 0.926236
x(1476) = -0.381957
x(1477) = -0.657307
x(1478) = -0.321949
x(1479) = -0.456720
x(1480) = 0.185865
x(1481) = 0.429288
x(1482) = 0.408380
x(1483) = 0.422549
x(1484) = -0.074007
x(1485) = 0.234695
x(1486) = 0.122212
x(1487) = -0.108629
x(1488) = -0.450253
x(1489) = -0.221148
x(1490) = -0.836013
x(1491) = 0.278500
x(1492) = 0.091948
x(1493) = -0.555759
x(1494) = -0.417649
x(1495) = -0.337340
x(1496) = 0.195292
x(1497) = -0.625589
x(1498) = -0.259554
x(1499) = 0.773289
```

Рисунок 3. – Скриншот фрагмента результата выполнения задания 2

Время выполнения прямого хода метода Гаусса:



```
Консоль отладки Microsoft Visual Studio
Time is:: 1.045942 sec
C:\Users\kaslon\Desktop\IPS\Task_3\task_3_1\Release\Task.exe (процесс 13516) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
```

Рисунок 4. – Скриншот результата выполнения задания 2

Задание 3: С помощью инструмента Amplifier XE определите наиболее часто используемые участки кода новой версии программы. Сохраните скриншот результатов анализа Amplifier XE. Создайте, на основе последовательной функции *SerialGaussMethod()*, новую функцию, реализующую параллельный метод Гаусса. Введите параллелизм в новую функцию, используя *cilk_for*. *Примечание: произвести параллелизацию одного внутреннего цикла прямого хода метода Гаусса (определить какого именно), и внутреннего цикла обратного хода. Время выполнения по-прежнему измерять только для прямого хода.*

1) С помощью инструмента Amplifier XE определите наиболее часто используемые участки кода новой версии программы. Сохраните скриншот результатов анализа Amplifier XE

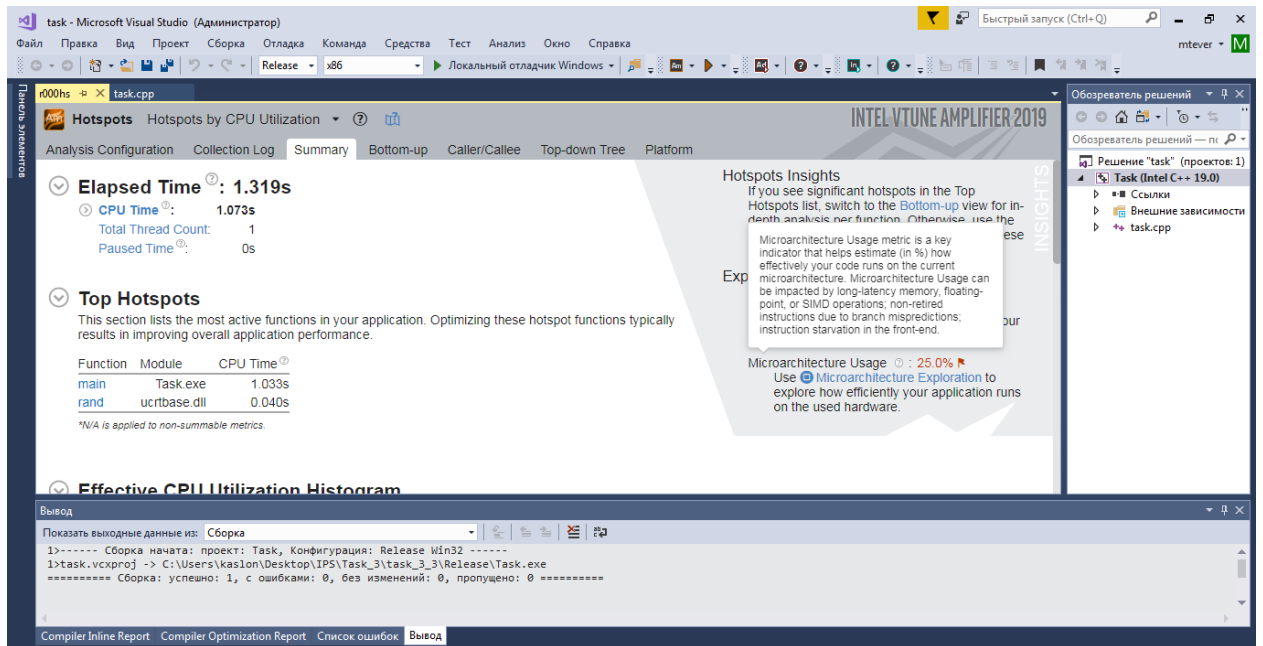


Рисунок 5. – Скриншот результата выполнения задания 3

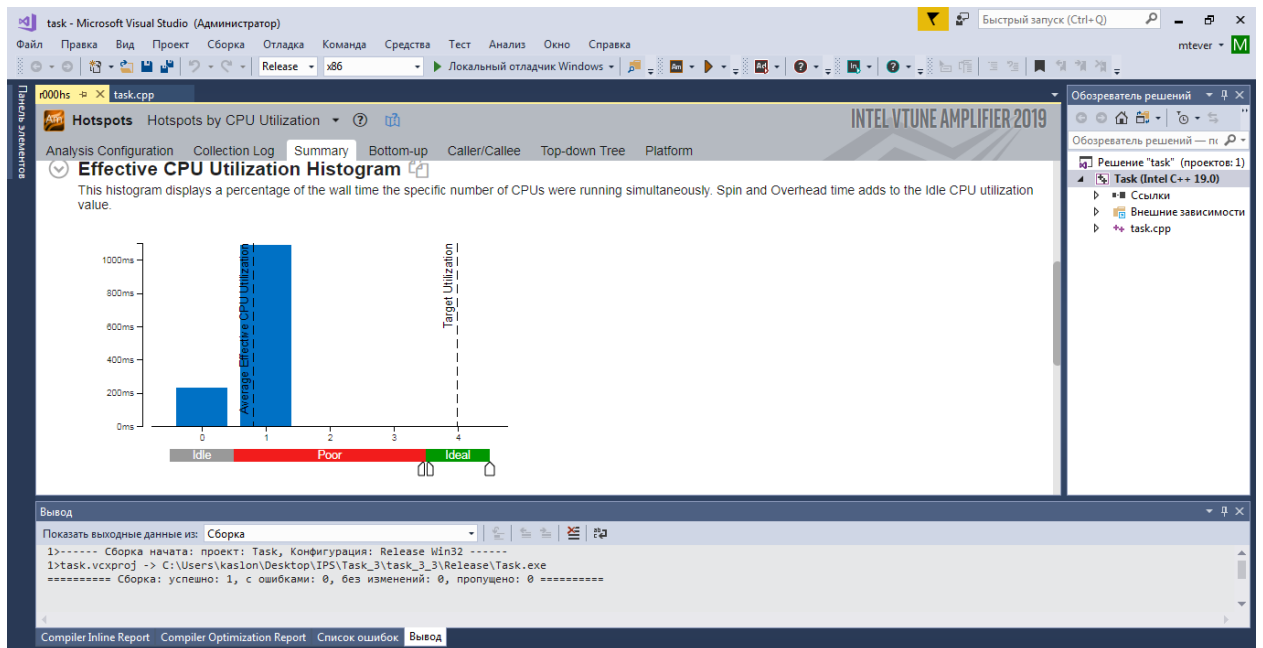


Рисунок 6. – Скриншот результата выполнения задания 3

2) Создайте, на основе последовательной функции *SerialGaussMethod()*, новую функцию, реализующую параллельный метод Гаусса. Введите параллелизм в новую функцию, используя *cilk_for*. Примечание: произвести параллелизацию одного внутреннего цикла прямого хода метода Гаусса (определить какого именно), и внутреннего цикла обратного хода. Время выполнения по-прежнему измерять только для прямого хода.

Код для прямого метода Гаусса с параллелизмом и измерением времени:

```

// прямой ход метода Гаусса
for (k = 0; k < rows; ++k)
{
  
```

```

//
for (int i = k + 1; i < rows; ++i)
{
    koef = -matrix[i][k] / matrix[k][k];

    cilk_for(int j = k; j <= rows; ++j)
    {
        matrix[i][j] += koef * matrix[k][j];
    }
}
}
t2 = high_resolution_clock::now();
duration<double> duration = (t2 - t1);
printf("Time is:: %lf sec\n\n", duration.count());

```

Код для обратного метода Гаусса с параллелизмом:

```

// обратный ход метода Гаусса
result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

for (k = rows - 2; k >= 0; --k)
{
    result[k] = matrix[k][rows];

    //
    cilk_for(int j = k + 1; j < rows; ++j)
    {
        result[k] -= matrix[k][j] * result[j];
    }

    result[k] /= matrix[k][k];
}

```

Ради интереса замерим скорость выполнения прямого метода Гаусса с параллелизмом:

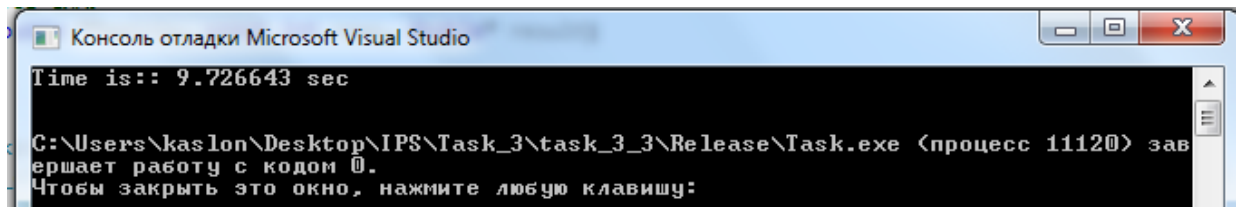


Рисунок 7. – Скриншот результата выполнения прямого метода Гаусса с параллелизмом

Анализ полученных результатов:

Время выполнения возросло. Посмотрим для понимания что к этому приводит в следующем задании с помощью *Inspector XE*.

Задание 4: Далее, используя *Inspector XE*, определите те данные (если таковые имеются), которые принимают участие в гонке данных или в других основных ошибках, возникающих при разработке параллельных программ, и устраните эти ошибки. Сохраните скриншоты анализов, проведенных инструментом *Inspector XE*: в случае обнаружения ошибок и после их устранения.

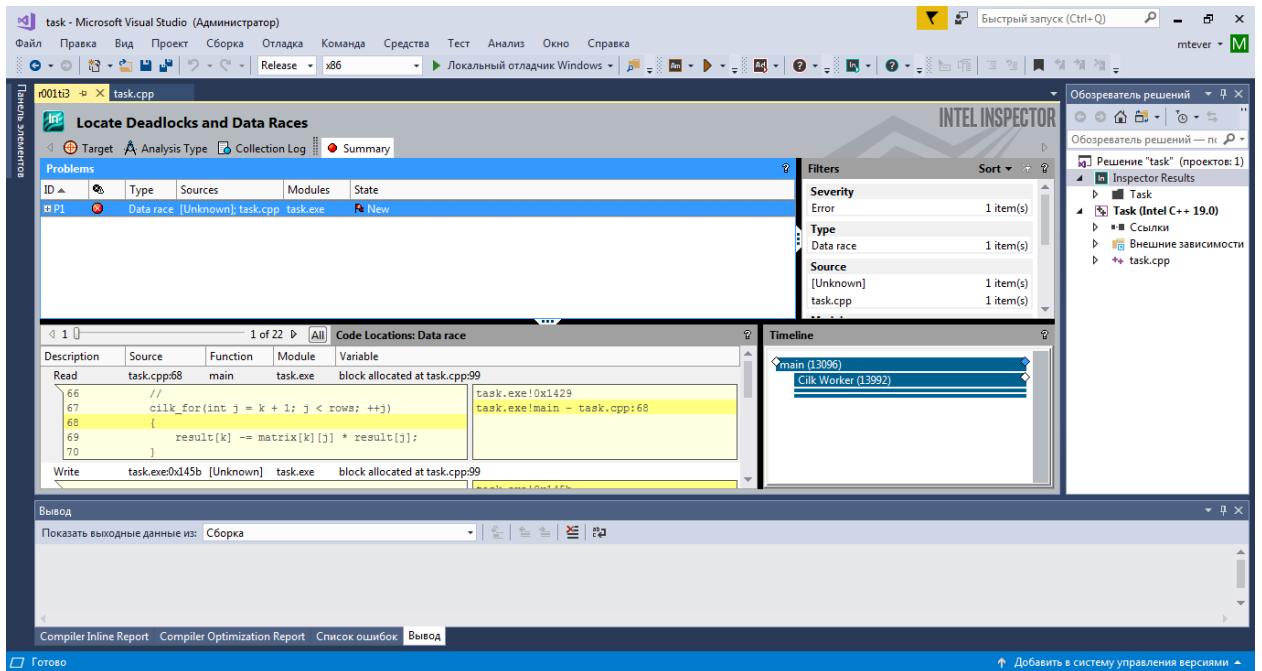


Рисунок 8. – Скриншот результата выполнения задания 4

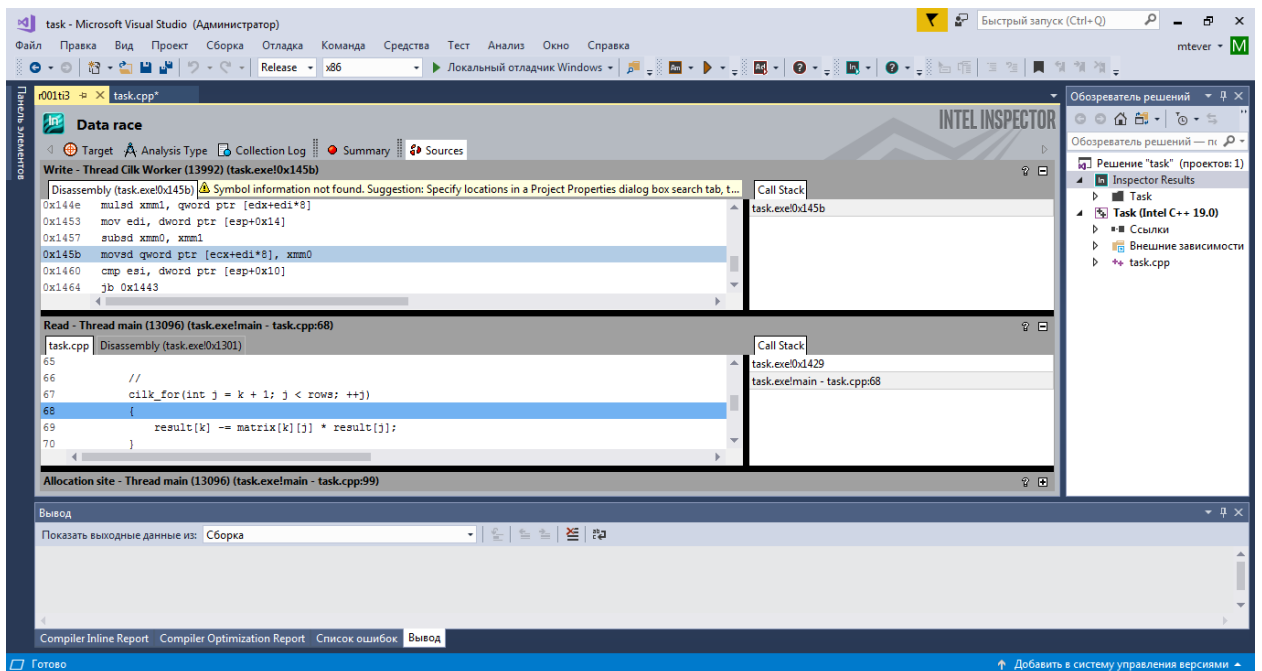


Рисунок 9. – Скриншот результата выполнения задания 4

Анализ полученных результатов:

Обнаружена гонка данных в обратном методе метода Гаусса. Попробуем исправить ошибки с помощью reduce.

Исправленный участок кода:

```
// обратный ход метода Гаусса
result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

for (k = rows - 2; k >= 0; --k)
{
```



```

        result[k] = matrix[k][rows];

        //
        cilk::reducer_opadd<double> result_k(matrix[k][rows]);
        cilk_for(int j = k + 1; j < rows; ++j)
        {
            result[k] -= matrix[k][j] * result[j];
        }

        result[k] /= matrix[k][k];
    }

```

Inspector XE после изменение кода:

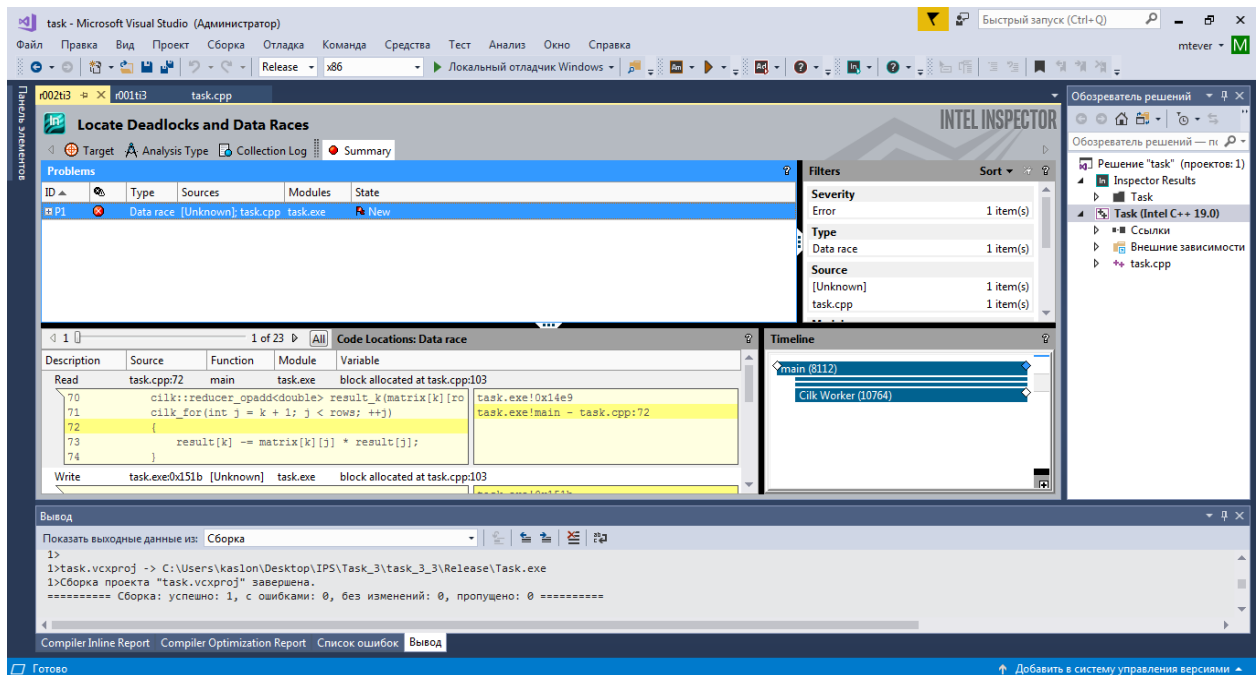
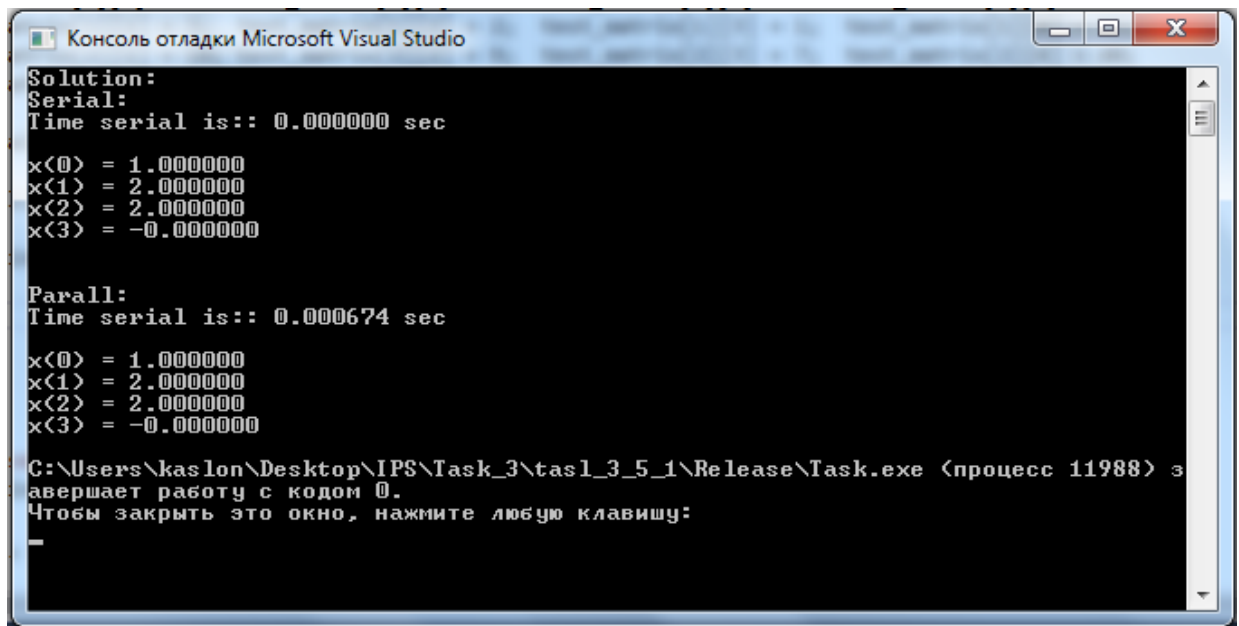


Рисунок 10. – Скриншот результата выполнения задания 4

Задание 5: Убедитесь на примере тестовой матрицы *test_matrix* в том, что функция, реализующая параллельный метод Гаусса работает правильно. Сравните время выполнения прямого хода метода Гаусса для последовательной и параллельной реализации при решении матрицы, имеющей количество строк *MATRIX_SIZE*, заполняющейся случайными числами. Запускайте проект в режиме Release, предварительно убедившись, что включена оптимизация (*Optimization->Optimization=/O2*). Подсчитайте ускорение параллельной версии в сравнении с последовательной. Выводите значения ускорения на консоль.

1) Убедитесь на примере тестовой матрицы *test_matrix* в том, что функция, реализующая параллельный метод Гаусса работает правильно

Проверим, что функция, реализующая параллельный метод Гаусса работает правильно:



```
Консоль отладки Microsoft Visual Studio
Solution:
Serial:
Time serial is:: 0.000000 sec
x(0) = 1.000000
x(1) = 2.000000
x(2) = 2.000000
x(3) = -0.000000

Parall:
Time serial is:: 0.000674 sec
x(0) = 1.000000
x(1) = 2.000000
x(2) = 2.000000
x(3) = -0.000000

C:\Users\kaslon\Desktop\IPS\Task_3\tas1_3_5_1\Release\Task.exe (процесс 11988) з
авершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

Рисунок 11. – Скриншот результата задания 5

Анализ полученных результатов:

Получили те же значения, код работает верно. Чуть больше затраченное время объясняется по результатам предыдущей лабораторной работы – при малых количествах данных параллельная реализация получается быстрее.

2) Сравните время выполнения прямого хода метода Гаусса для последовательной и параллельной реализации при решении матрицы, имеющей количество строк *MATRIX_SIZE*, заполняющейся случайными числами. Запускайте проект в режиме Release, предварительно убедившись, что включена оптимизация (*Optimization->Optimization=/O2*). Подсчитайте ускорение параллельной версии в сравнении с последовательной. Выводите значения ускорения на консоль.

Проверим, что включена оптимизация (*Optimization->Optimization=/O2*)

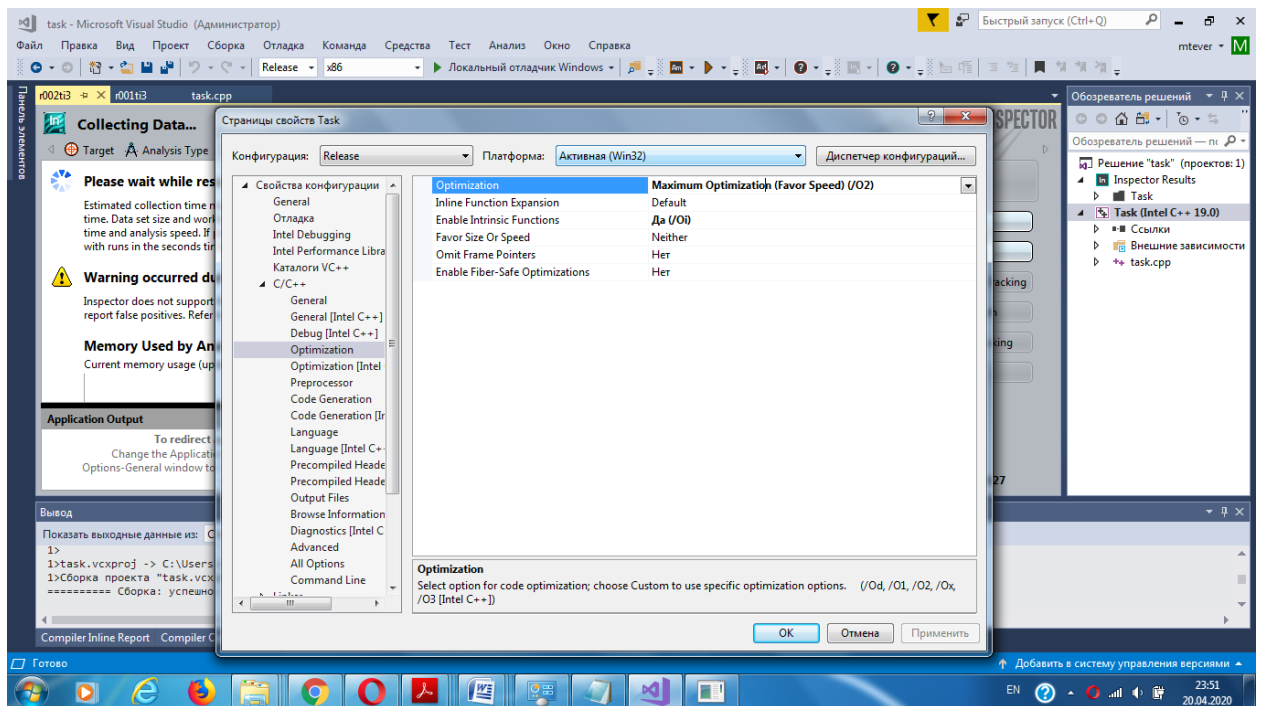


Рисунок 12. – Скриншот проверки настроек проекта

Код программы:

```
#include <stdio.h>
#include <ctime>
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#include <chrono>

using namespace std::chrono;

// количество строк в исходной квадратной матрице
const int MATRIX_SIZE = 1500;

/// Функция InitMatrix() заполняет переданную в качестве
/// параметра квадратную матрицу случайными значениями
/// matrix - исходная матрица СЛАУ
void InitMatrix(double** matrix)
{
    for (int i = 0; i < MATRIX_SIZE; ++i)
    {
        matrix[i] = new double[MATRIX_SIZE + 1];
    }

    for (int i = 0; i < MATRIX_SIZE; ++i)
    {
        for (int j = 0; j <= MATRIX_SIZE; ++j)
        {
            matrix[i][j] = rand() % 2500 + 1;
        }
    }
}

/// Функция SerialGaussMethod() решает СЛАУ методом Гаусса
/// matrix - исходная матрица коэффициентов уравнений, входящих в СЛАУ,
/// последний столбец матрицы - значения правых частей уравнений
/// rows - количество строк в исходной матрице
/// result - массив ответов СЛАУ
duration<double> SerialGaussMethod(double **matrix, const int rows, double* result)
```

```

{
    int k;
    double koef;
    high_resolution_clock::time_point t1, t2;
    t1 = high_resolution_clock::now();
    // прямой ход метода Гаусса
    for (k = 0; k < rows; ++k)
    {
        //
        for (int i = k + 1; i < rows; ++i)
        {
            koef = -matrix[i][k] / matrix[k][k];

            for (int j = k; j <= rows; ++j)
            {
                matrix[i][j] += koef * matrix[k][j];
            }
        }
    }
    //
    t2 = high_resolution_clock::now();
    t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);

    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

    for (k = rows - 2; k >= 0; --k)
    {
        result[k] = matrix[k][rows];

        //
        for (int j = k + 1; j < rows; ++j)
        {
            result[k] -= matrix[k][j] * result[j];
        }

        result[k] /= matrix[k][k];
    }
    return duration;
}

duration<double> ParallGaussMethod(double **matrix, const int rows, double* result)
{
    int k;
    high_resolution_clock::time_point start = high_resolution_clock::now();
    // прямой ход метода Гаусса
    for (k = 0; k < rows; ++k)
    {
        cilk_for(int i = k + 1; i < rows; ++i)
        {
            double koef = -matrix[i][k] / matrix[k][k];

            for (int j = k; j <= rows; ++j)
            {
                matrix[i][j] += koef * matrix[k][j];
            }
        }
    }
    high_resolution_clock::time_point finish = high_resolution_clock::now();
    duration<double> duration = (finish - start);
    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];
}

```

```

    for (k = rows - 2; k >= 0; --k)
    {
        cilk::reducer_opadd<double> result_k(matrix[k][rows]);

        cilk_for(int j = k + 1; j < rows; ++j)
        {
            result_k -= matrix[k][j] * result[j];
        }

        result[k] = result_k->get_value() / matrix[k][k];
    }

    return duration;
}

int main()
{
    srand((unsigned)time(0));

    int i;

    // кол-во строк в матрице, приводимой в качестве примера
    //const int test_matrix_lines = 4;
    //Меняем на:
    const int test_matrix_lines = MATRIX_SIZE;

    double **test_matrix = new double*[test_matrix_lines];

    // цикл по строкам
    for (i = 0; i < test_matrix_lines; ++i)
    {
        // (test_matrix_lines + 1)- количество столбцов в тестовой матрице,
        // последний столбец матрицы отведен под правые части уравнений, входящих в
        СЛАУ
        test_matrix[i] = new double[test_matrix_lines + 1];

        // массив решений СЛАУ
        double *result = new double[test_matrix_lines];
        double *result_parall = new double[test_matrix_lines];

        // инициализация тестовой матрицы
        /*test_matrix[0][0] = 2; test_matrix[0][1] = 5; test_matrix[0][2] = 4;
        test_matrix[0][3] = 1; test_matrix[0][4] = 20;
        test_matrix[1][0] = 1; test_matrix[1][1] = 3; test_matrix[1][2] = 2;
        test_matrix[1][3] = 1; test_matrix[1][4] = 11;
        test_matrix[2][0] = 2; test_matrix[2][1] = 10; test_matrix[2][2] = 9;
        test_matrix[2][3] = 7; test_matrix[2][4] = 40;
        test_matrix[3][0] = 3; test_matrix[3][1] = 8; test_matrix[3][2] = 9;
        test_matrix[3][3] = 2; test_matrix[3][4] = 37; */
        //меняем на:
        InitMatrix(test_matrix);

        duration<double>duration_serial = SerialGaussMethod(test_matrix,
        test_matrix_lines, result);

        duration<double>duration_parall = ParallGaussMethod(test_matrix,
        test_matrix_lines, result_parall);

        for (i = 0; i < test_matrix_lines; ++i)
        {
            delete[]test_matrix[i];

```

```

    }

    printf("Solution:\n");
    printf("Serial:\n");
    printf("Time serial is:: %lf sec\n\n", duration_serial.count());
    for (i = 0; i < test_matrix_lines; ++i)
    {
        //printf("x(%d) = %lf\n", i, result[i]);
    }

    printf("\n\nParall:\n");
    printf("Time serial is:: %lf sec\n\n", duration_parall.count());
    for (i = 0; i < test_matrix_lines; ++i)
    {
        //printf("x(%d) = %lf\n", i, result_parall[i]);
    }

    printf("Boost is:: %lf\n\n", duration_parall.count()/ duration_serial.count());

    delete[] result;
    delete[] result_parall;

    return 0;
}

```

Результат:

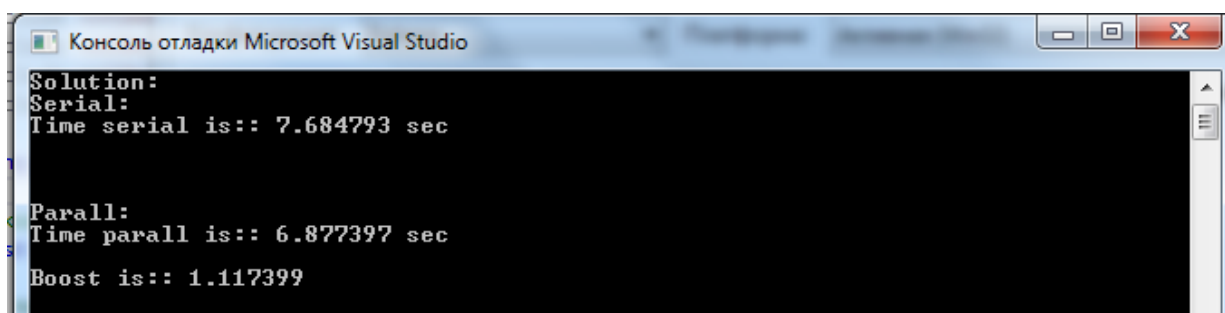


Рисунок 13. – Скриншот результатов задания 5

Анализ полученных результатов:

Использование параллелизма для такого большого массива данных (1500) позволило решить СЛАУ методом Гаусса в 1.117399 раз быстрее. Или примерно на 0.807 секунды быстрее.

Выводы:

В рамках задания к занятию 3:

- 1) Разобран код, реализующий метод Гаусса для решения СЛАУ и проверен в Matlab
- 2) Были повторены такие надстройки и функции, как:
 - 1) Amplifier XE
 - 2) Inspector XE
 - 3) Вычисление времени выполнения программы при помощи библиотеки «chrono»
 - 4) reducer_opadd
- 3) Были рассмотрены скорости выполнения программы решения СЛАУ методом Гаусса последовательной и параллельной программой

4) На базе полученных результатов были сделаны выводы