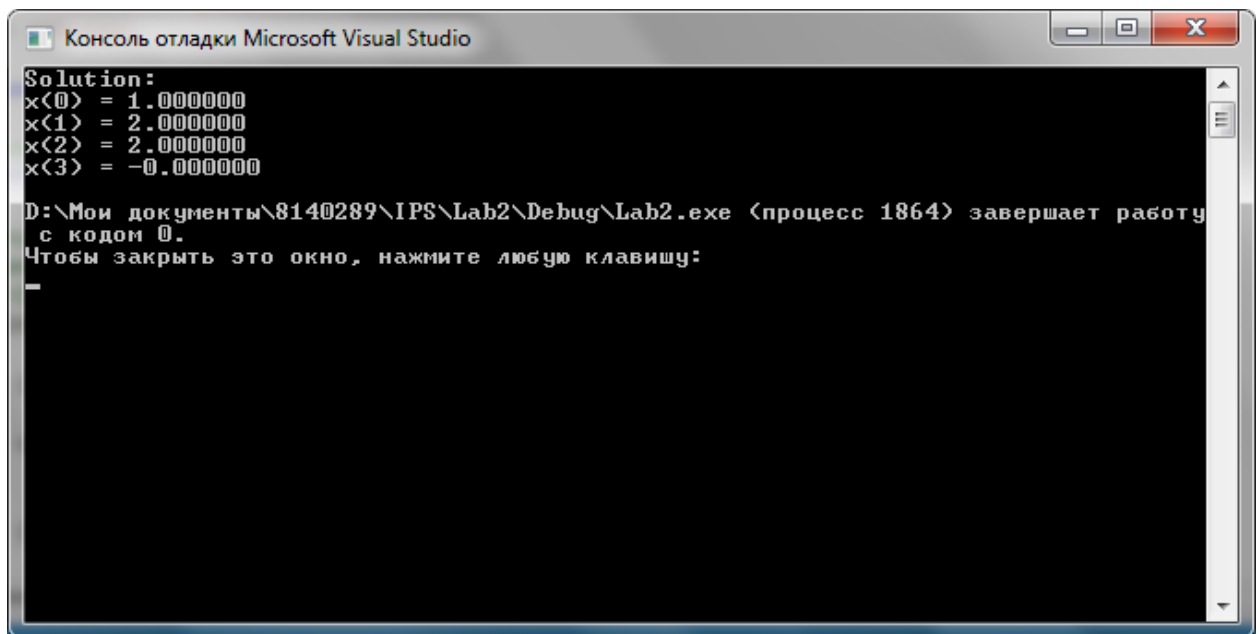


Отчет к занятию №3

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

Запуск первоначальной версии программы:



```
Консоль отладки Microsoft Visual Studio

Solution:
x<0> = 1.000000
x<1> = 2.000000
x<2> = 2.000000
x<3> = -0.000000

D:\Мои документы\8140289\IPS\Lab2\Debug\Lab2.exe (процесс 1864) завершает работу
с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

Проверим правильность решения в Matlab:

```
>> M = [2 5 4 1 20;
1 3 2 1 11;
2 10 9 7 40;
3 8 9 2 37];
>> x = M(:,1:end-1)^(-1)*M(:,end)

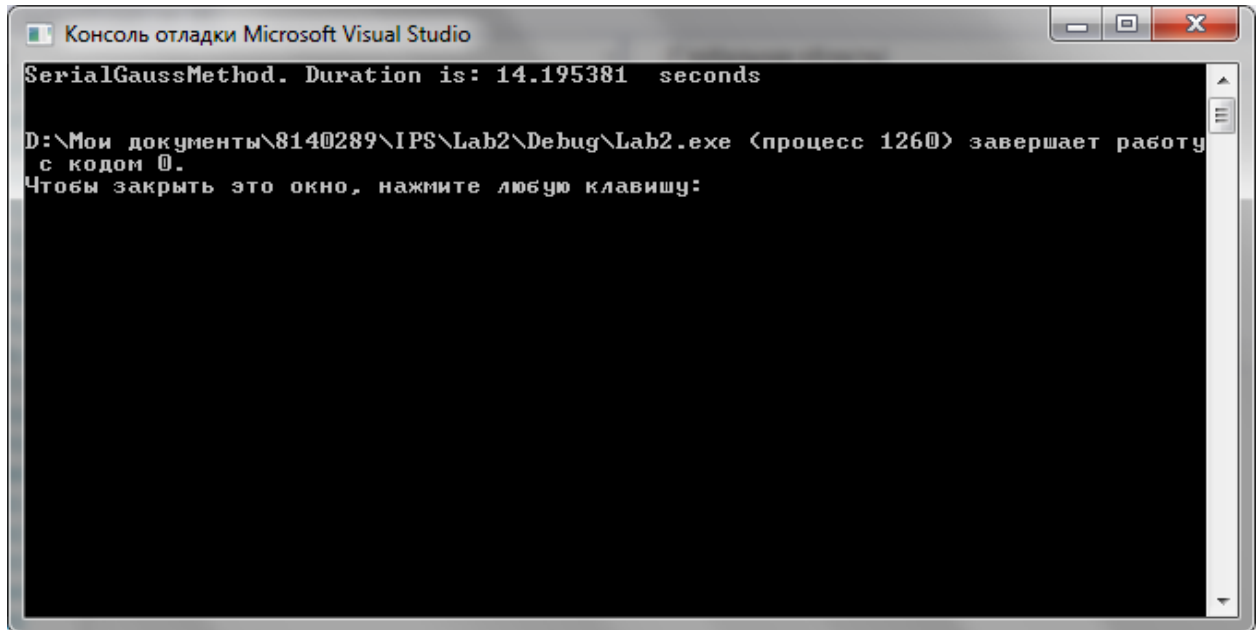
x =

1.0000
2.0000
2.0000
-0.0000
```

Решение найдено верно.

Определим время работы для последовательного метода гаусса для матрицы с количеством строк **MATRIX_SIZE**:

(убрали вывод решения)



```
Консоль отладки Microsoft Visual Studio
SerialGaussMethod. Duration is: 14.195381 seconds
D:\Мои документы\8140289\IPS\Lab2\Debug\Lab2.exe (процесс 1260) завершает работу
с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
```

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.

Hotspots Hotspots by CPU Utilization ?

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform

Elapsed Time[?]: 6.217s

CPU Time[?]: 5.295s
 Total Thread Count: 1
 Paused Time[?]: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time [?]
SerialGaussMethod	IPS1.exe	5.076s
rand	ucrtbased.dll	0.200s
free_dbg	ucrtbased.dll	0.010s
malloc	ucrtbased.dll	0.009s

*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

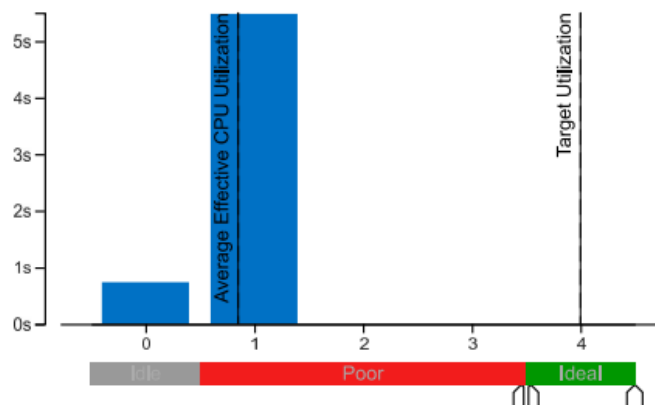
Explore Additional Insights

Parallelism[?]: 21.3%
 Use Threading to explore more opportunities to increase parallelism in your application.

INSIGHTS

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Создадим функцию, реализующую параллельный метод Гаусса:

```

/// Функция ParallelGaussMethod() решает СЛАУ параллельным методом Гаусса
void ParallelGaussMethod(double** matrix, const int rows, double* result)
{
    int k;
    double koef;

    high_resolution_clock::time_point 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];
            cilk_for (int j = k; j <= rows; ++j)
            {

```

```

        matrix[i][j] += koef * matrix[k][j];
    }

    }

    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
    printf("ParallelGaussMethod. Duration is: %lf seconds \n \n", duration.count());

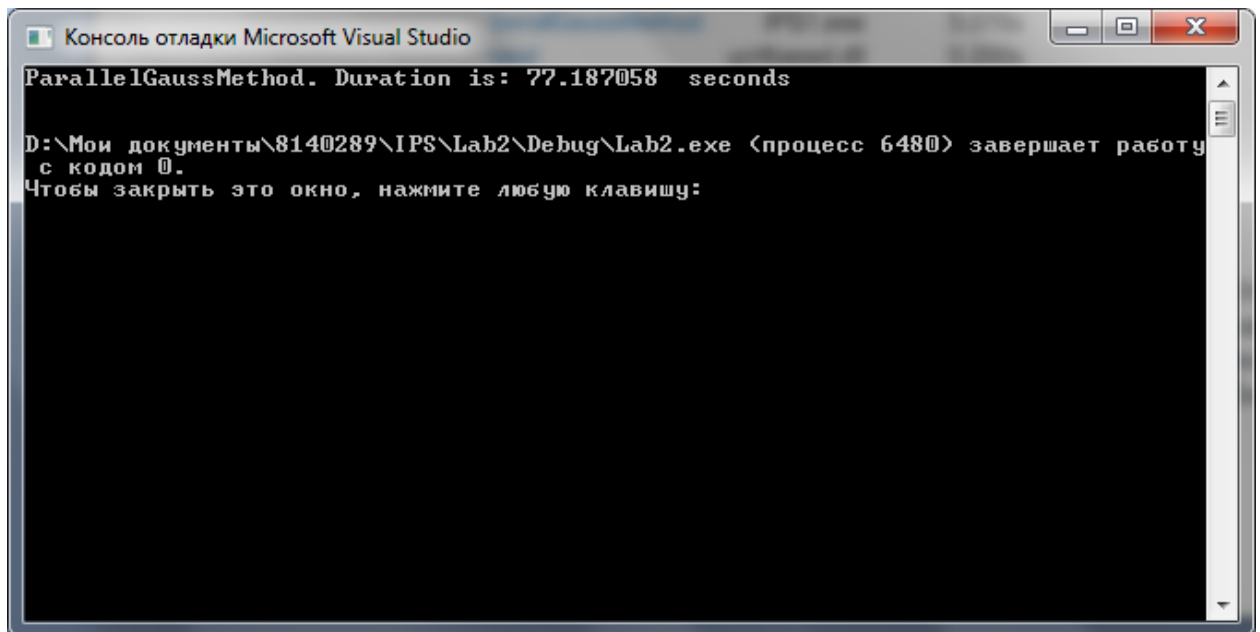
    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

    for (k = rows - 2; k >= 0; --k)
    {
        result[k] = matrix[k][rows];
        cilk_for(int j = k + 1; j < rows; ++j)
        {
            result[k] -= matrix[k][j] * result[j];
        }

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

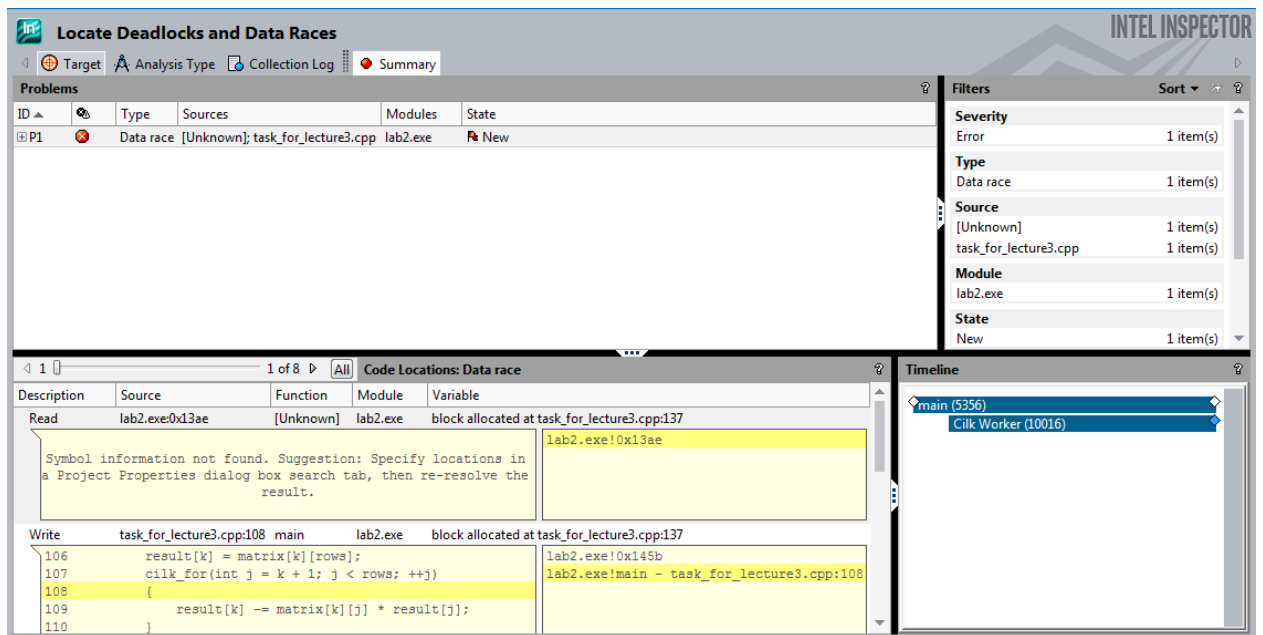
```

Определим время выполнения:



Видно, что метод стал работать еще медленнее, что неправильно. Используем **Inspector XE** для поиска ошибок.

- 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.



Обнаружена гонка данных.

Новая функция:

```

/// Функция ParallelGaussMethod() решает СЛАУ параллельным методом Гаусса
void ParallelGaussMethod(double** matrix, const int rows, double* result)
{
    int k;

    high_resolution_clock::time_point t1 = 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 t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
    printf("ParallelGaussMethod. Duration is: %lf seconds \n \n", duration.count());

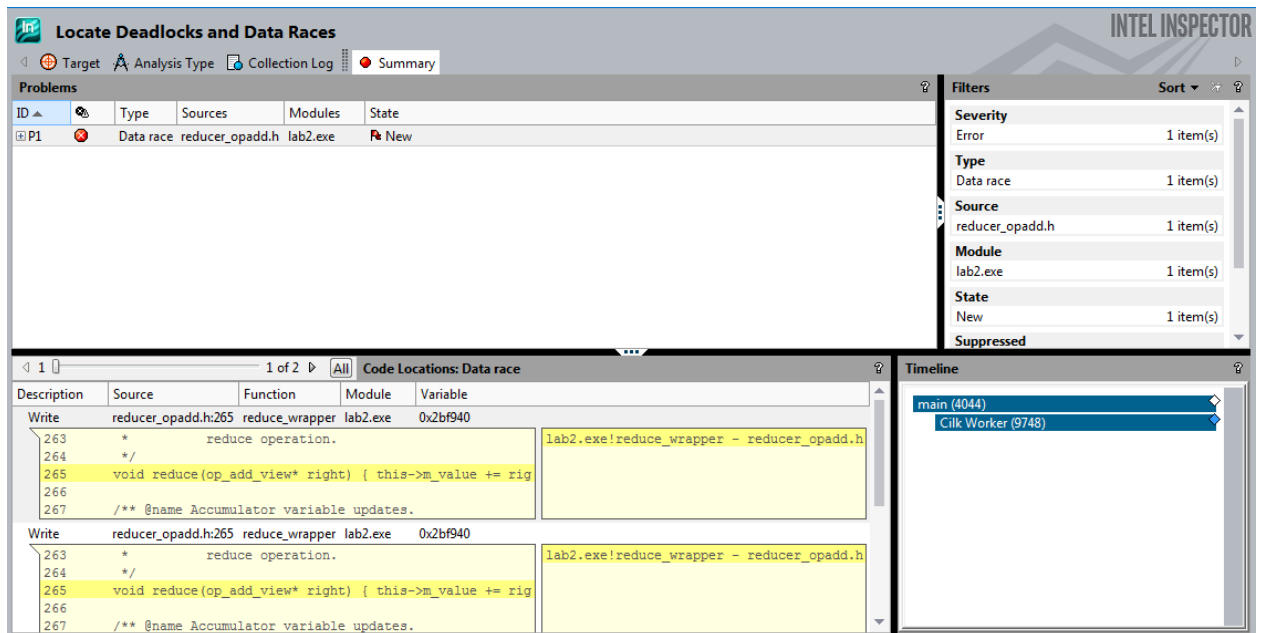
    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

    for (k = rows - 2; k >= 0; --k)
    {
        cilk::reducer_opadd<double> res(matrix[k][rows]);
        cilk_for(int j = k + 1; j < rows; ++j)
        {
            res -= matrix[k][j] * result[j];
        }

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

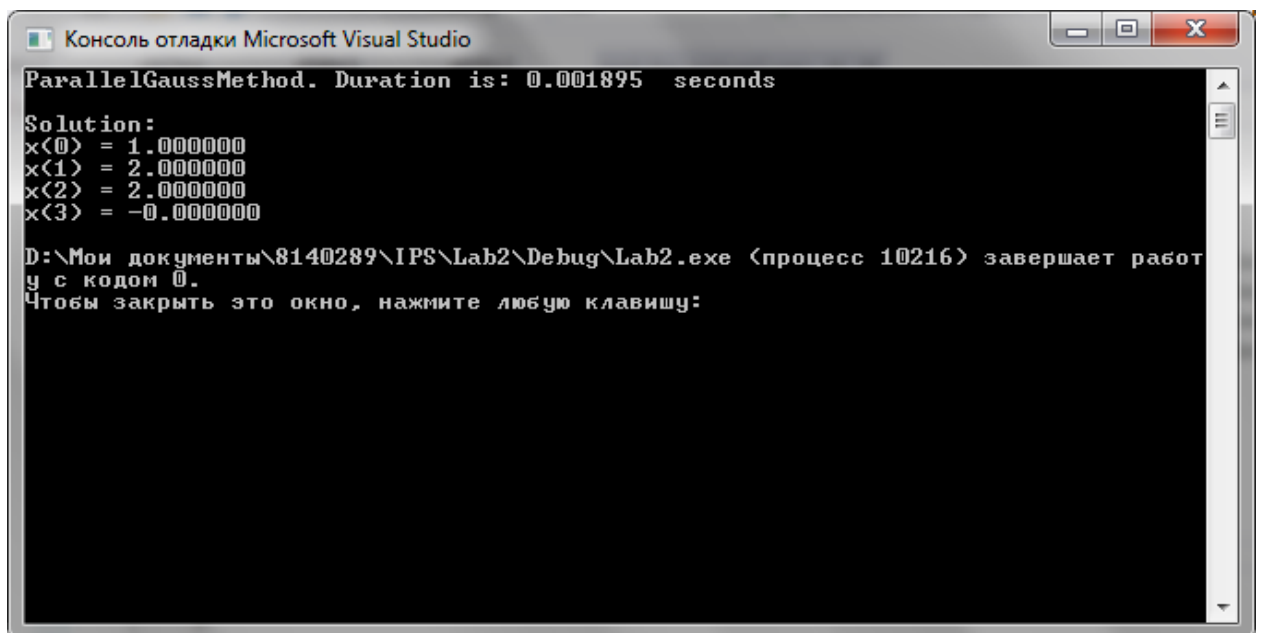
```

Анализ после устранения ошибок:



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.

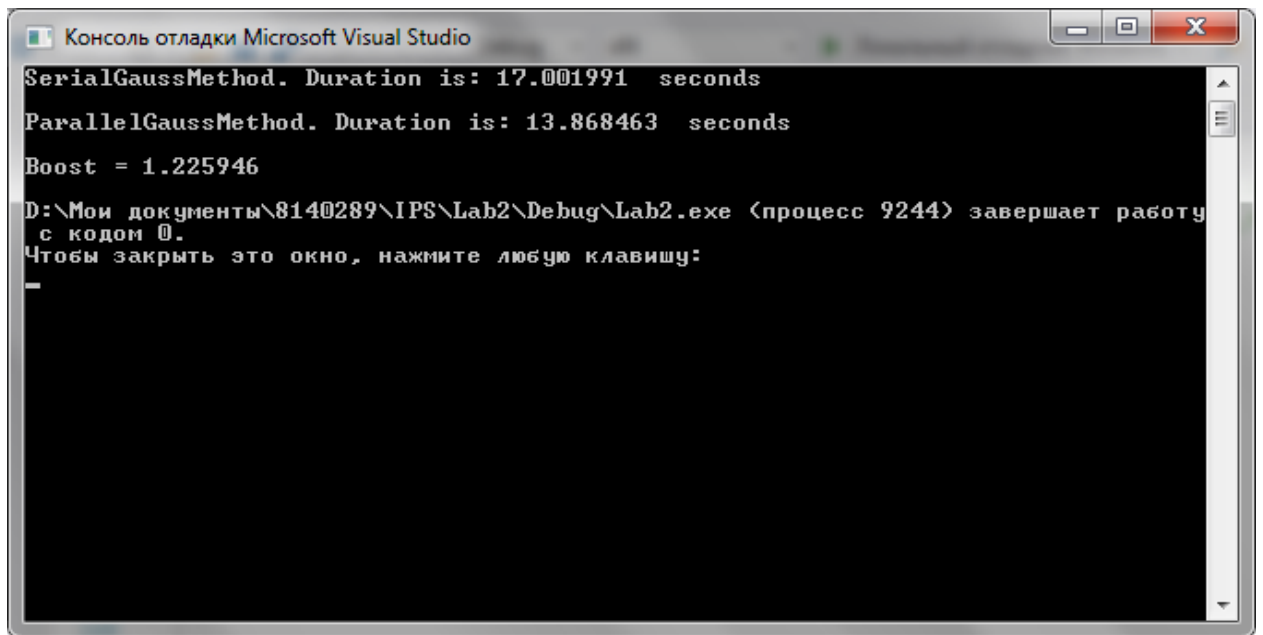
Поиск решения параллельным методом Гаусса для тестовой матрицы:



Метод работает верно.

Сравнение времени для последовательного и параллельного методов:

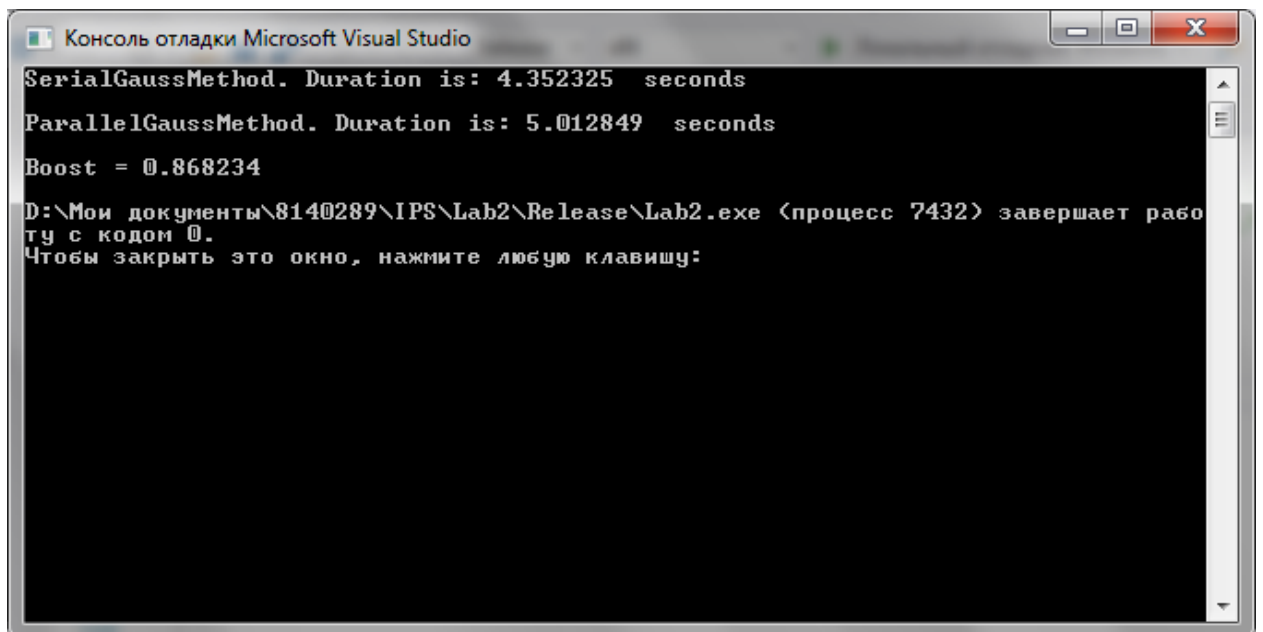
(в режиме Debug):



Консоль отладки Microsoft Visual Studio

```
SerialGaussMethod. Duration is: 17.001991 seconds  
ParallelGaussMethod. Duration is: 13.868463 seconds  
Boost = 1.225946  
D:\Мои документы\8140289\IPS\Lab2\Debug\Lab2.exe <процесс 9244> завершает работу  
с кодом 0.  
Чтобы закрыть это окно, нажмите любую клавишу:  
—
```

(в режиме Release с включенной оптимизацией):



Консоль отладки Microsoft Visual Studio

```
SerialGaussMethod. Duration is: 4.352325 seconds  
ParallelGaussMethod. Duration is: 5.012849 seconds  
Boost = 0.868234  
D:\Мои документы\8140289\IPS\Lab2\Release\Lab2.exe <процесс 7432> завершает рабо  
ту с кодом 0.  
Чтобы закрыть это окно, нажмите любую клавишу:
```