
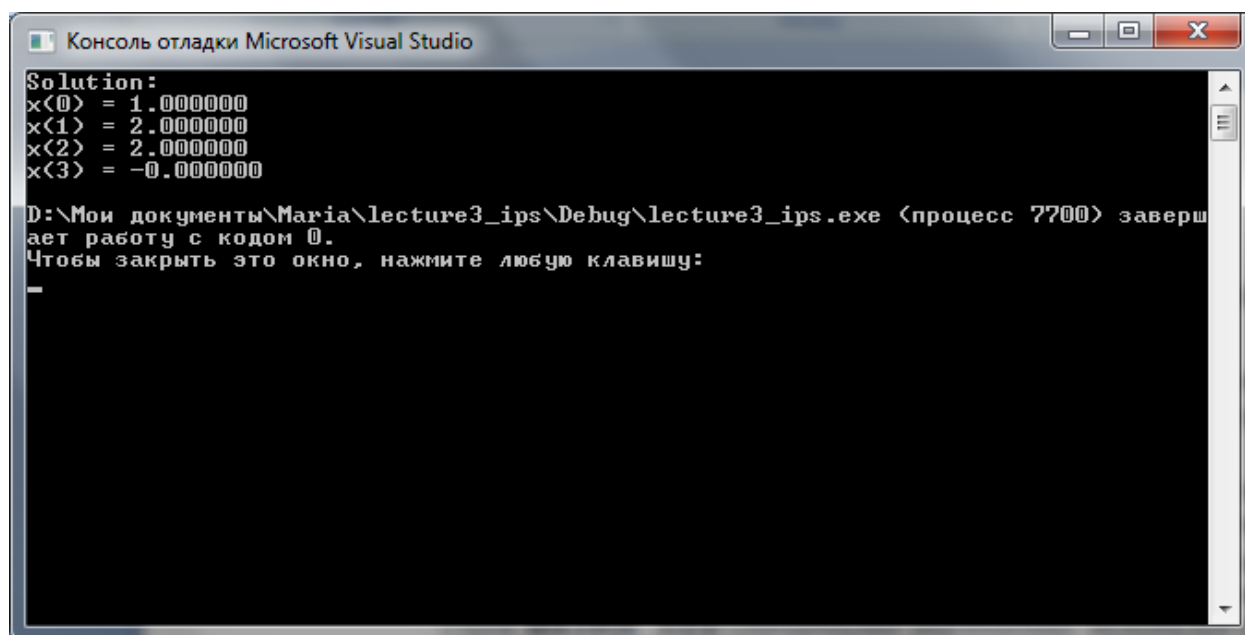


Задание к занятию №3

Выполнила: Шахманова Мария, ПМ-21м

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**).

Найдём решение для тестовой матрицы:



```
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\Maria\lecture3_ips\Debug\lecture3_ips.exe (процесс 7700) заверш
ает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
—
```

Проверим в Matlab, что $A \cdot X = B$ (значения B - в последнем столбце матрицы A):

```
A = [2 5 4 1 20; 1 3 2 1 11; 2 10 9 7 40; 3 8 9 2 37];
X = [1 2 2 0];
eq(A(:, 1:4) * X', A(:, 5))
ans =
     1
     1
     1
     1
```

Т.о., найденное решение – верное.

Найдём время выполнения и первые 15 решений для матрицы с числом строк MATRIX_SIZE:

```
Консоль отладки Microsoft Visual Studio
Duration for direct evaluation in gauss method: 13.938992
Solution:
x<0> = -1.747538
x<1> = -1.391563
x<2> = 0.484137
x<3> = 0.599507
x<4> = -0.884179
x<5> = -1.023445
x<6> = -2.385268
x<7> = -1.283389
x<8> = 0.017939
x<9> = 2.802490
x<10> = 0.405947
x<11> = 1.812622
x<12> = -3.122914
x<13> = 1.884750
x<14> = -2.662303
D:\Мои документы\Maria\lecture3_ips\Debug\lecture3_ips.exe (процесс 5804) заверш
ает 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** ? : 6.334s

⌵ **CPU Time** ? : 4.658s

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	4.428s
rand	ucrtbased.dll	0.148s
_stdio_common_vfprintf	ucrtbased.dll	0.041s
InitMatrix	IPS1.exe	0.040s
main	IPS1.exe	0.000s

*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

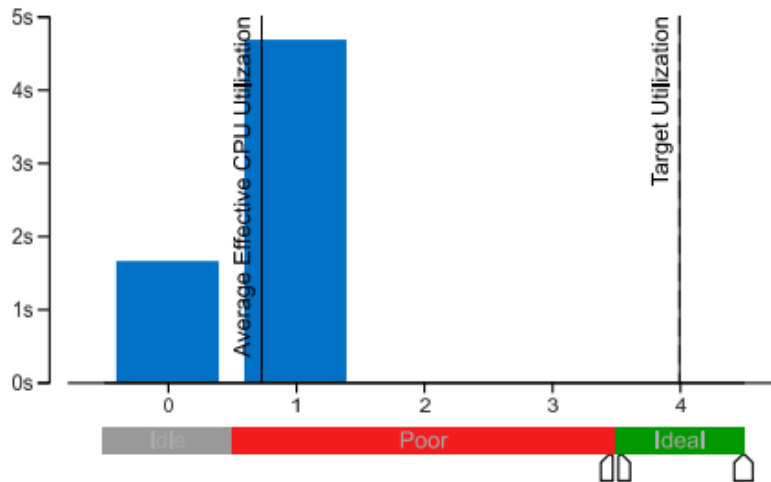
Parallelism ? : 18.4% ⬆

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.



Реализуем параллельный метод 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)
        {
            double 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_for = (t2 - t1);
    printf("Duration for direct evaluation in gauss method: %lf\n\n", duration_for);

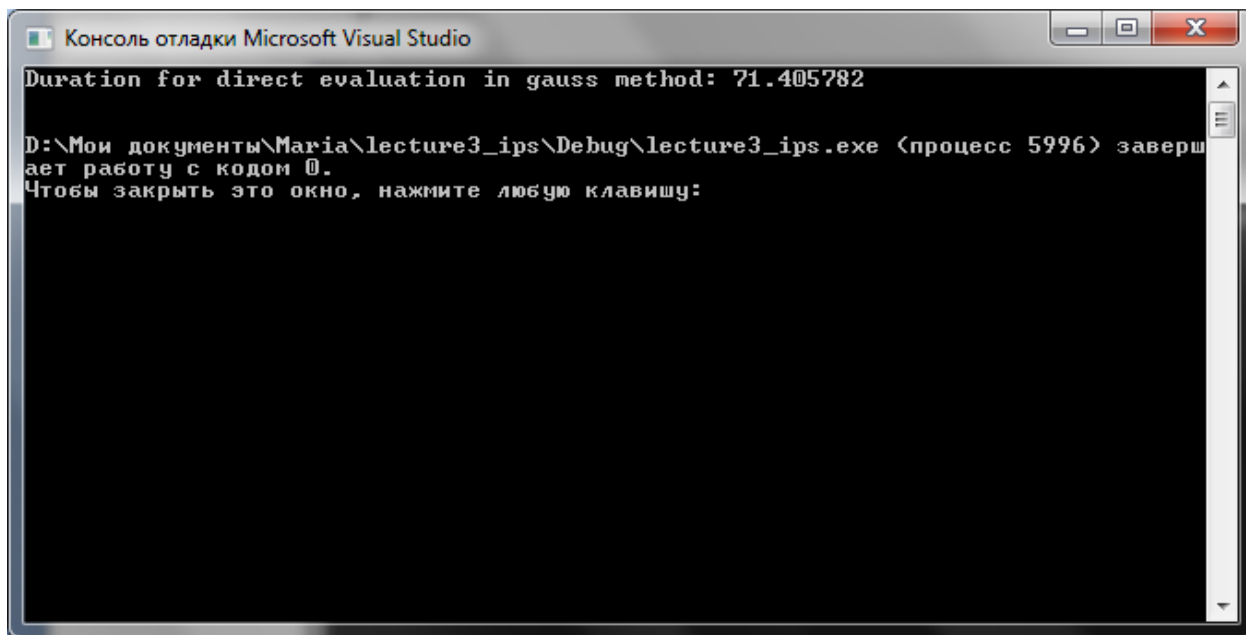
    // обратный ход метода Гаусса
    result[rows - 1] = matrix[rows - 1][rows] / matrix[rows - 1][rows - 1];

    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];
        }

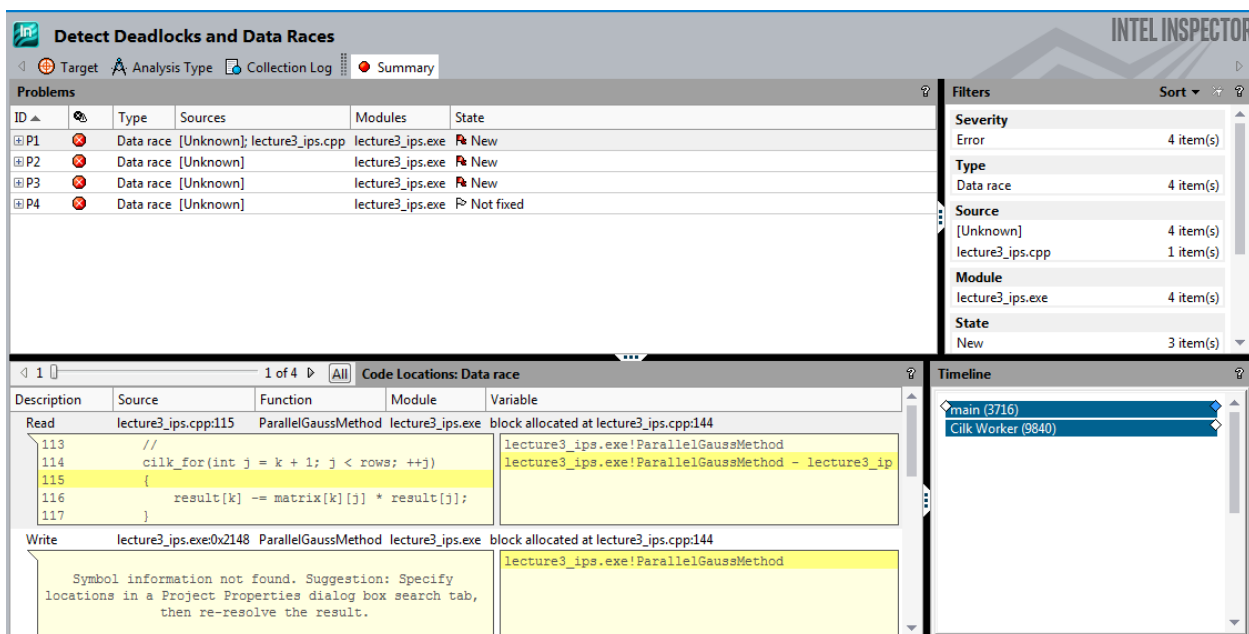
        result[k] /= matrix[k][k];
    }
}
```

Запустим программу и проанализируем результаты:



Как видно, время выполнения увеличилось, а значит, наш метод реализован неверно. Для выяснения возможных причин, перейдём к п.4 и воспользуемся Inspector XE.

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.



В цикле для `result[k] -= matrix[k][j] * result[j]`; много потоков считают `matrix[k][j] * result[j]` и пытаются выполнить `-=` с `result[k]`. Из-за чего происходит гонка данных.

Устраним ошибки и проанализируем код ещё раз:

```
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)
{
    //
    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_for_par = (t2 - t1);
printf("Duration for direct evaluation in Parallel gauss method: %lf\n\n", duration_for_par);

// 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_(matrix[k][rows]);
    //result[k] = matrix[k][rows];

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

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

```

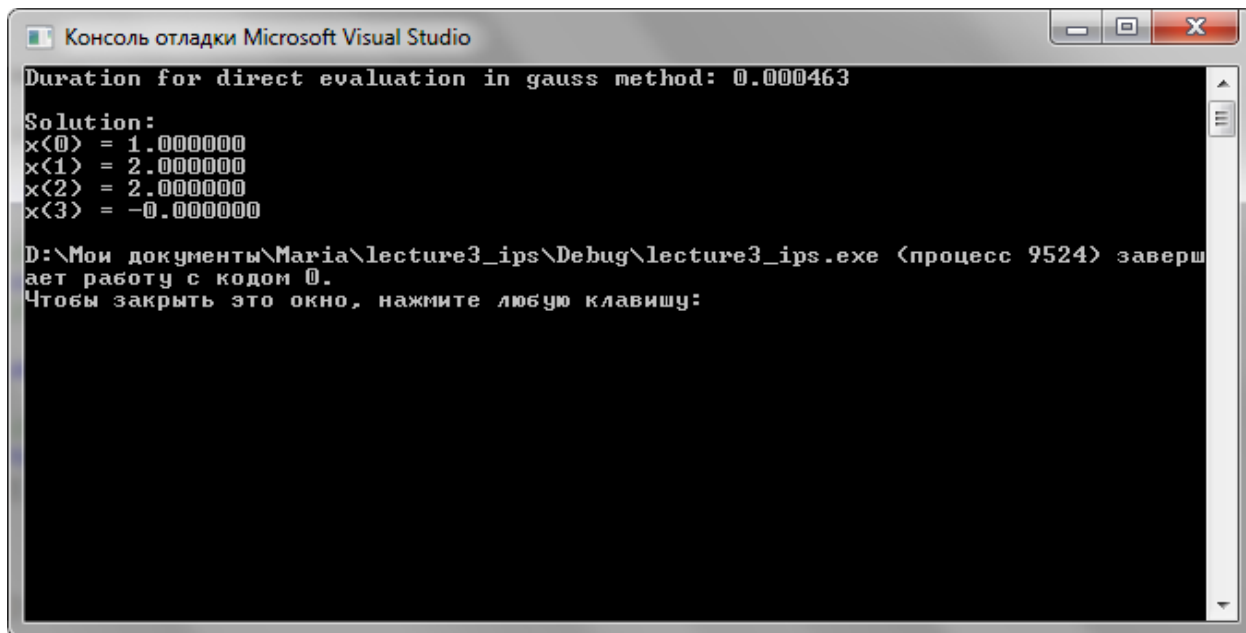
The screenshot shows the Intel Inspector interface with the title "Detect Deadlocks and Data Races". The "Problems" pane on the left lists a single issue: "Data race" in "reducer.h: reducer_opadd.h" within the module "lecture3_ips.exe", marked as "New". The "Filters" pane on the right shows the issue details: Severity (Error), Type (Data race), Source (reducer.h, reducer_opadd.h), Module (lecture3_ips.exe), and State (New). The main pane displays "Code Locations: Data race" with two entries:

Description	Source	Function	Module	Variable
Write	reducer_opadd.h:280	operator-=	lecture3_ips.exe	0x579500
<pre> 278 /** Decrements the accumulator variable by @a x. 279 */ 280 op_add_views operator-=(const Type& x) { this->m_value 281 282 /** Pre-increment. </pre>				
Write	reducer.h:201	allocate	lecture3_ips.exe	0x579500
<pre> 199 * @return An untyped pointer to the allocated me 200 */ 201 void* allocate(size_t s) const { return operator new(s 202 203 /** Deallocates raw memory pointed to by @a p </pre>				

The "Timeline" pane on the right shows two threads: "main (9592)" and "Cilk Worker (6172)".

Гонки данных не происходит.

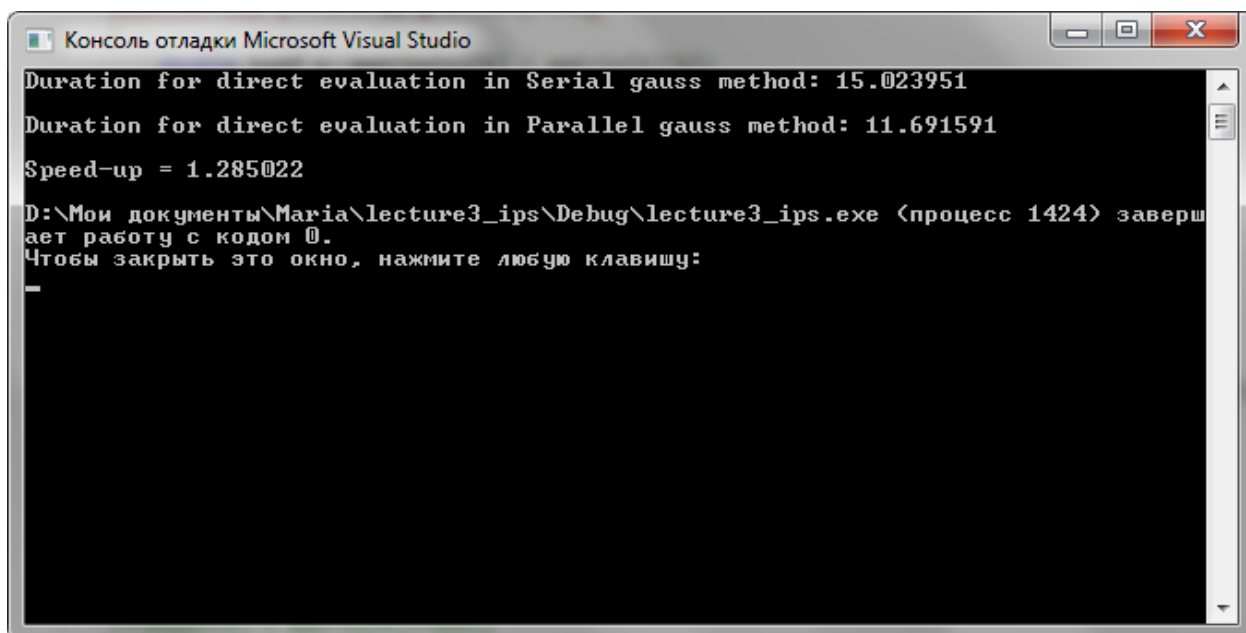
5. Убедитесь на примере тестовой матрицы **test_matrix** в том, что функция, реализующая параллельный метод Гаусса работает правильно. Сравните время выполнения прямого хода метода Гаусса для последовательной и параллельной реализации при решении матрицы, имеющей количество строк **MATRIX_SIZE**, заполняющейся 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.



```
Консоль отладки Microsoft Visual Studio
Duration for direct evaluation in gauss method: 0.000463
Solution:
x(0) = 1.000000
x(1) = 2.000000
x(2) = 2.000000
x(3) = -0.000000
D:\Мои документы\Maria\lecture3_ips\Debug\lecture3_ips.exe (процесс 9524) заверш
ает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
```

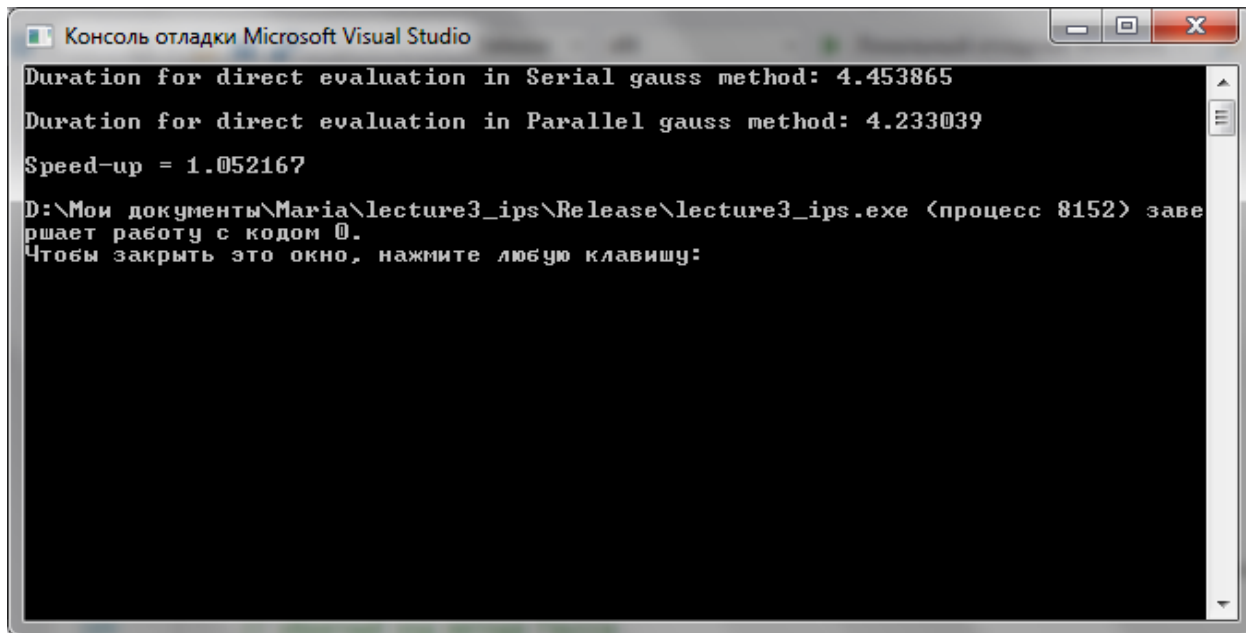
Результаты аналогичны п.1. Т.е. метод работает верно.

Сравним результаты двух методов. В режиме Debug:



```
Консоль отладки Microsoft Visual Studio
Duration for direct evaluation in Serial gauss method: 15.023951
Duration for direct evaluation in Parallel gauss method: 11.691591
Speed-up = 1.285022
D:\Мои документы\Maria\lecture3_ips\Debug\lecture3_ips.exe (процесс 1424) заверш
ает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

В режиме Release:



```
Консоль отладки Microsoft Visual Studio
Duration for direct evaluation in Serial gauss method: 4.453865
Duration for direct evaluation in Parallel gauss method: 4.233039
Speed-up = 1.052167
D:\Мои документы\Maria\lecture3_ips\Release\lecture3_ips.exe (процесс 8152) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
```

Вывод: параллельный метод привёл к небольшому ускорению. (В режиме Debug заметно сильнее).