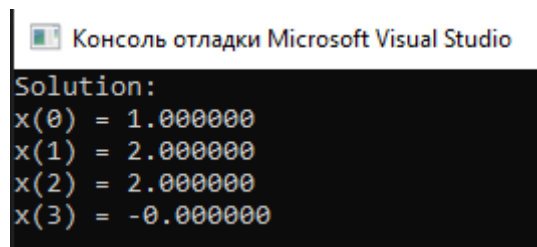


1. В файле `task_for_lecture3.cpp` приведен код, реализующий последовательную версию метода Гаусса для решения СЛАУ. Проанализируйте представленную программу.
2. Запустите первоначальную версию программы и получите решение для тестовой матрицы `test_matrix`, убедитесь в правильности приведенного алгоритма. Добавьте строки кода для измерения времени выполнения прямого хода метода Гаусса в функцию `SerialGaussMethod()`. Заполните матрицу количеством строк `MATRIX_SIZE` случайными значениями, используя функцию `InitMatrix()`. Найдите решение СЛАУ для этой матрицы. (закомментируйте строки кода, где используется тестовая матрица `test_matrix`).

Запустим первоначальную версию программы



```

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

Проверим правильность полученного решения

Количество неизвестных величин в системе:

**Изменить названия переменных в системе**

Заполните систему линейных уравнений:

$$\begin{cases} 2x_1 + 5x_2 + 4x_3 + 1x_4 = 20 \\ 1x_1 + 3x_2 + 2x_3 + 1x_4 = 11 \\ 2x_1 + 10x_2 + 9x_3 + 7x_4 = 40 \\ 3x_1 + 8x_2 + 9x_3 + 2x_4 = 37 \end{cases}$$

**Ответ:**

$$\begin{cases} x_1 = 1 \\ x_2 = 2 \\ x_3 = 2 \\ x_4 = 0 \end{cases}$$

Как видно, на тестовых данных алгоритм работает правильно. Добавим строки кода для измерения времени выполнения прямого хода метода Гаусса

```

/// 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 start, finish;
    start = 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];
            }
        }
    }

    finish = high_resolution_clock::now();
    duration<double> duration = (finish - start);
    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];
    }
}

```

Найдем решение СЛАУ для матрицы, заполненной случайными значениями.

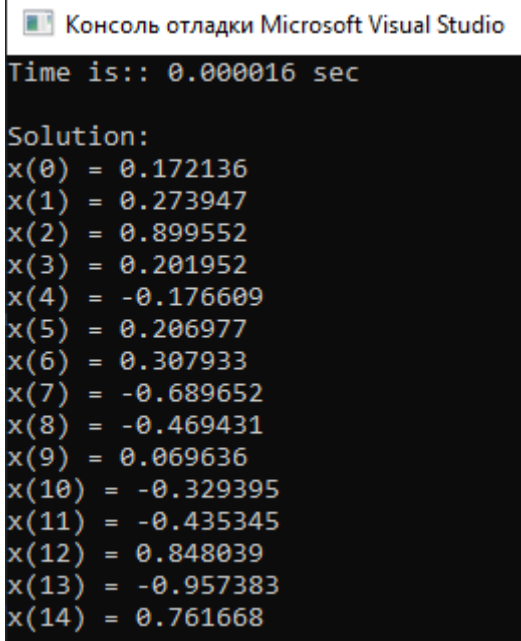
```

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

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

InitMatrix(test_matrix);
// массив решений СЛАУ
double *result = new double[test_matrix_lines];
SerialGaussMethod(test_matrix, test_matrix_lines, result);

```



```

Консоль отладки Microsoft Visual Studio

Time is:: 0.000016 sec

Solution:
x(0) = 0.172136
x(1) = 0.273947
x(2) = 0.899552
x(3) = 0.201952
x(4) = -0.176609
x(5) = 0.206977
x(6) = 0.307933
x(7) = -0.689652
x(8) = -0.469431
x(9) = 0.069636
x(10) = -0.329395
x(11) = -0.435345
x(12) = 0.848039
x(13) = -0.957383
x(14) = 0.761668

```

3. С помощью инструмента Amplifier XE определите наиболее часто используемые участки кода новой версии программы. Создайте на основе последовательной функции, новую функцию, используя `cilk_for`.

```

/// Функция ParallelGaussMethod() 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 ParallelGaussMethod(double **matrix, const int rows, double* result)
{
    int k;
    double koef;

    high_resolution_clock::time_point start, finish;
    start = 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];
            }
        }
    }

    finish = high_resolution_clock::now();
    duration<double> duration = (finish - start);
    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];
    cilk_for (int j = k + 1; j < rows; ++j)
    {
        result[k] -= matrix[k][j] * result[j];
    }

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

```

4. Далее, используя Inspector XE, определите те данные, которые принимают участие в гонке данных или в других основных ошибках, возникающий при разработке параллельных программ, и устраните эти ошибки.

**Locate Deadlocks and Data Races**

Target Analysis Type Collection Log Summary

ID	Type	Sources	Modules	State
P1	Data race [Unknown]		ips1.exe	New
P2	Data race [Unknown]		ips1.exe	Not fixed
P3	Data race [Unknown]		ips1.exe	New
P4	Data race [Unknown]		ips1.exe	Not fixed
P5	Data race [Unknown]; main.cpp	ips1.exe	ips1.exe	New

Code Locations: Data race

Description	Source	Function	Module	Variable
Write	main.cpp:119	ParallelGaussMethod	ips1.exe	block allocated at main.cpp:149
<pre> 117 // 118 cilk_for (int j = k + 1; j &lt; rows; ++j) 119 { 120     result[k] -= matrix[k][j] * result[j]; 121 } </pre>				
Write	ips1.exe:0x22e8	ParallelGaussMethod	ips1.exe	block allocated at main.cpp:149
<pre> ips1.exe!ParallelGaussMethod </pre>				

Была найдена гонка данных. Исправим ошибки.

```

/// Функция ParallelGaussMethod() 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 ParallelGaussMethod(double **matrix, const int rows, double* result)
{
    int k;
    high_resolution_clock::time_point start, finish;
    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];
        }
    }
}
finish = high_resolution_clock::now();
duration<double> duration_parallel = (finish - start);
printf("Time is:: %lf sec\n\n", duration_parallel.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> 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];
}
}

```

Анализ после устранения ошибок.

**Intel Inspector**

**Locate Deadlocks and Data Races**

Target: Analysis Type: Collection Log: Summary

**Problems**

ID	Type	Sources	Modules	State
P1	Data race	reducer.h; reducer_opadd.h	ips1.exe	New

**Filters**

Severity	Count
Error	1 item(s)

**Type**

Type	Count
Data race	1 item(s)

**Source**

Source	Count
reducer.h	1 item(s)
reducer_opadd.h	1 item(s)

**Module**

Module	Count
ips1.exe	1 item(s)

**State**

State	Count
New	1 item(s)

**Code Locations: Data race**

Description	Source	Function	Module	Variable
Write	reducer_opadd.h:280	operator-=	ips1.exe	0xb5d100
<pre> 278 /** Decrements the accumulator variable by @a x. 279 */ 280 op_add_view&amp; operator-=(const Type&amp; x) { this-&gt;m_val 281 282 /** Pre-increment. </pre>				
Read	ips1.exe!operator-=			
<pre> ips1.exe!operator-= - reducer_opadd.h:28 ips1.exe!operator-= - reducer_opadd.h:43 ips1.exe!ParallelGaussMethod </pre>				
Write	reducer.h:201	allocate	ips1.exe	0xb5d100
<pre> 199 * @return An untyped pointer to the allocated 200 */ 201 void* allocate(size_t s) const { return operator new 202 </pre>				
Read	ips1.exe!allocate			
<pre> ips1.exe!allocate - reducer.h:201 ips1.exe!allocate_wrapper - reducer.h:94 ips1.exe!view - reducer.h:890 ips1.exe!view - reducer.h:1232 </pre>				

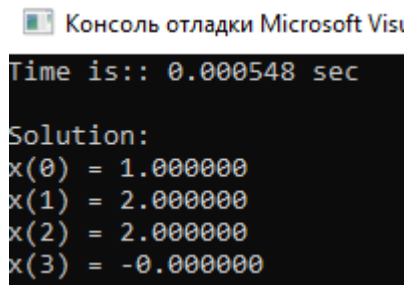
**Timeline**

Cilk Worker (9044)

Cilk Worker (9120)

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.

Убедимся на примере тестовой матрицы в правильности работы параллельного алгоритма

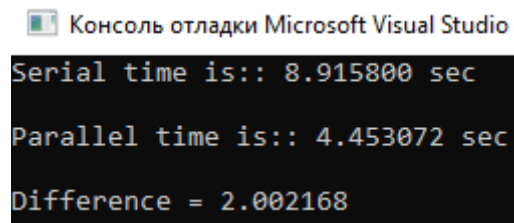


Консоль отладки Microsoft Visual Studio

```
Time is:: 0.000548 sec  
  
Solution:  
x(0) = 1.000000  
x(1) = 2.000000  
x(2) = 2.000000  
x(3) = -0.000000
```

MATRIX\_SIZE = 1500

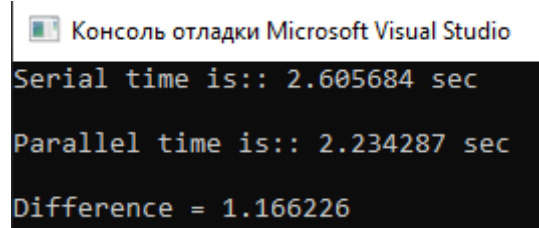
With optimization/Od



Консоль отладки Microsoft Visual Studio

```
Serial time is:: 8.915800 sec  
Parallel time is:: 4.453072 sec  
Difference = 2.002168
```

With optimization/O2



Консоль отладки Microsoft Visual Studio

```
Serial time is:: 2.605684 sec  
Parallel time is:: 2.234287 sec  
Difference = 1.166226
```