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).*

Запустим программу для тестовой матрицы. Получим:

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

Проверим результаты. В задании система уравнений имеет следующий вид:

$$\begin{cases} 2x_0 + 5x_1 + 4x_2 + x_3 = 20 \\ x_0 + 3x_1 + 2x_2 + x_3 = 11 \\ 2x_0 + 10x_1 + 9x_2 + 7x_3 = 40 \\ 3x_0 + 8x_1 + 9x_2 + 2x_3 = 37 \end{cases}$$

Программа выдала её решение:

$$\begin{cases} x_0 = 1 \\ x_1 = 2 \\ x_2 = 2 \\ x_3 = 0 \end{cases}$$

Подставив это решение в левую часть уравнения, мы должны получить правую часть системы:

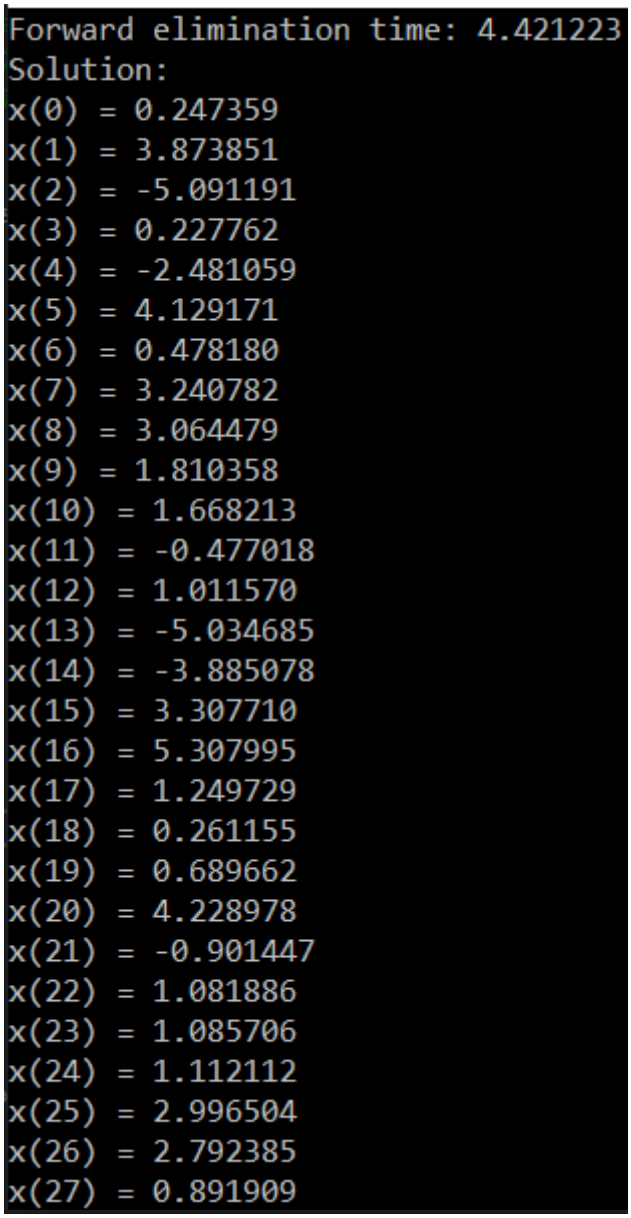
$$\begin{cases} 2 \cdot 1 + 5 \cdot 2 + 4 \cdot 2 + 0 = 20 \\ 1 + 3 \cdot 2 + 2 \cdot 2 + 0 = 11 \\ 2 \cdot 1 + 10 \cdot 2 + 9 \cdot 2 + 7 \cdot 0 = 40 \\ 3 \cdot 1 + 8 \cdot 2 + 9 \cdot 2 + 2 \cdot 0 = 37 \end{cases}$$

Мы получили правую часть системы, значит программа работает правильно.

Добавим замер времени:

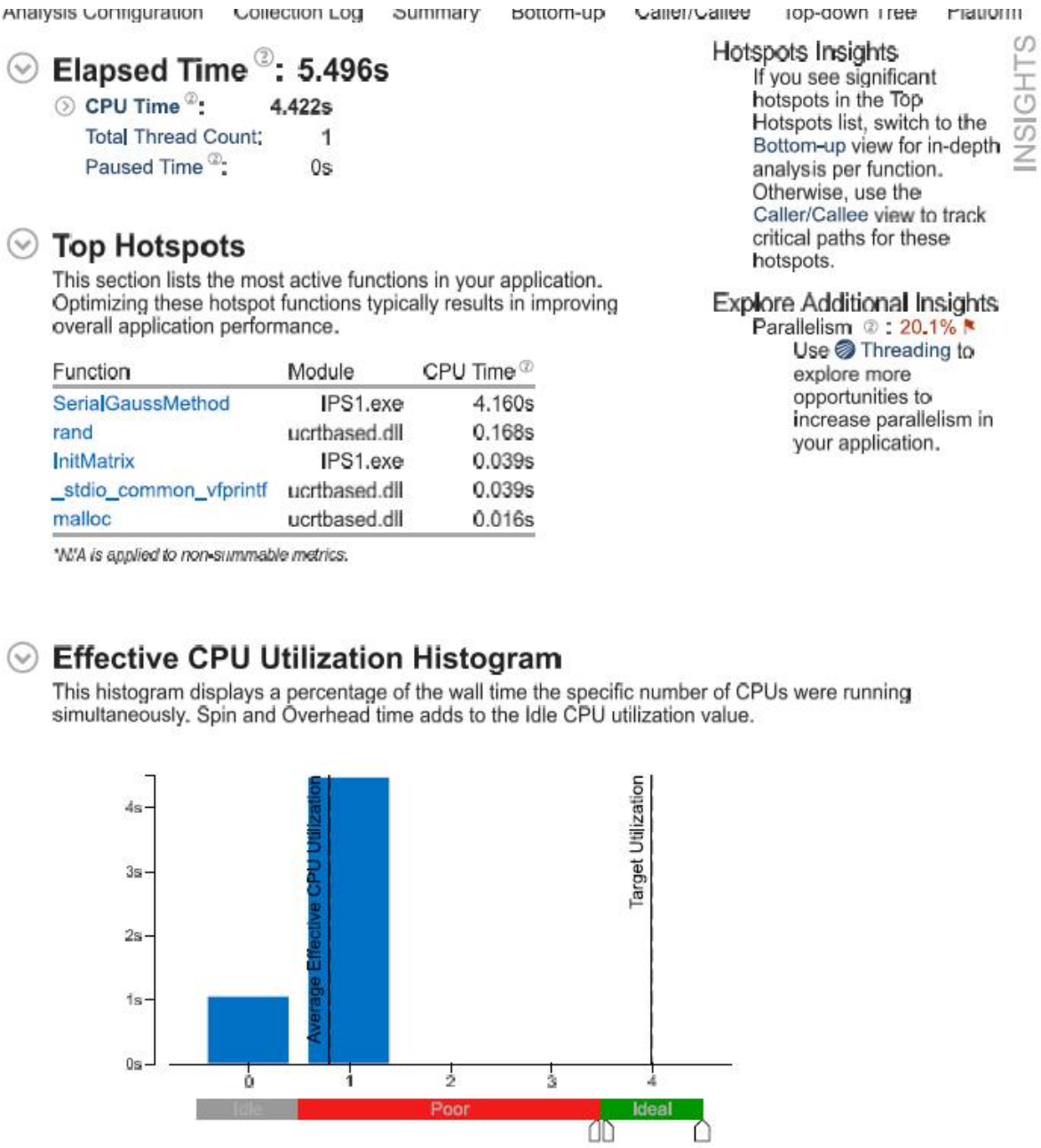
```
void SerialGaussMethod(double **matrix, const int rows, double* result)
{
	int k;
	double koef;

	auto t0 = 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];
			}
		}
	}
	auto t1 = high_resolution_clock::now();

	// 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];
	}

	duration<double> dur = t1 - t0;
	printf("Forward elimination time: %f\n", dur.count());
}
```

Найдем для нее решение:



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.*

Запустим Amplifier XE.



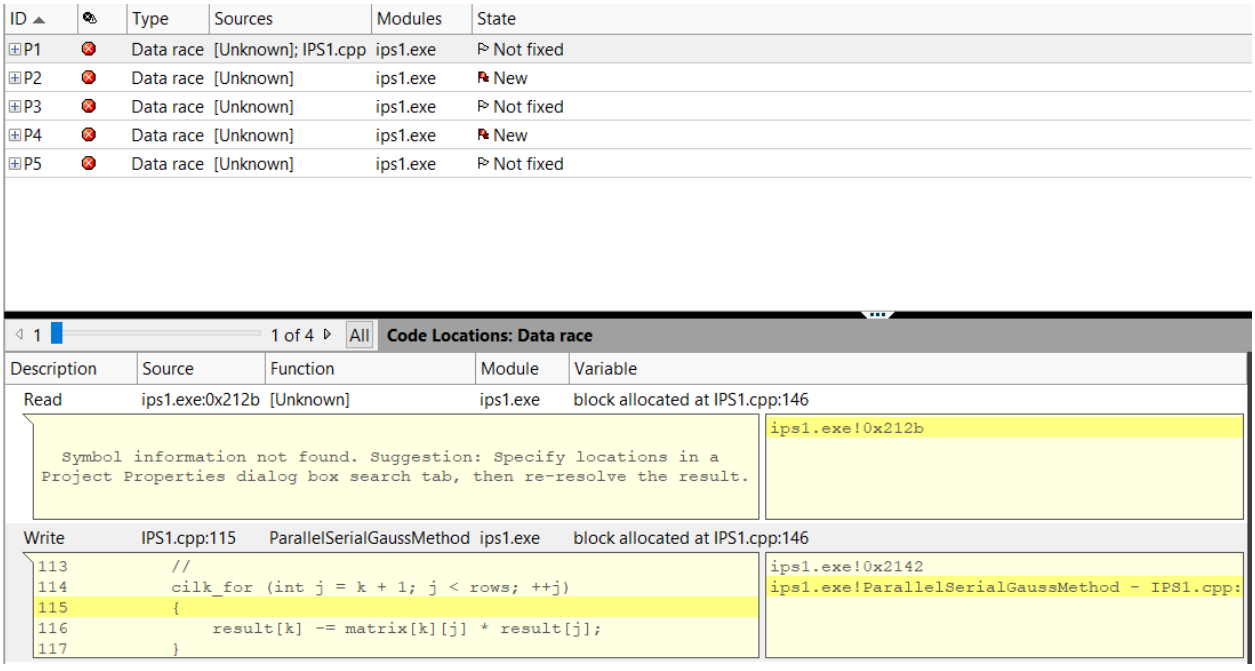
Видим, что функция SerialGaussMethod работает медленно, реализуем параллельную функцию.

```
void ParallelSerialGaussMethod(double **matrix, const int rows, double* result)
{
	int k;
	double koef;

	auto t0 = 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];
                        }
                }
        }
        auto t1 = high_resolution_clock::now();

        // 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];
        }

        duration<double> dur = t1 - t0;
        printf("Forward elimination time: %f\n", dur.count());
}
```

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.*

Запустим Inspector XE

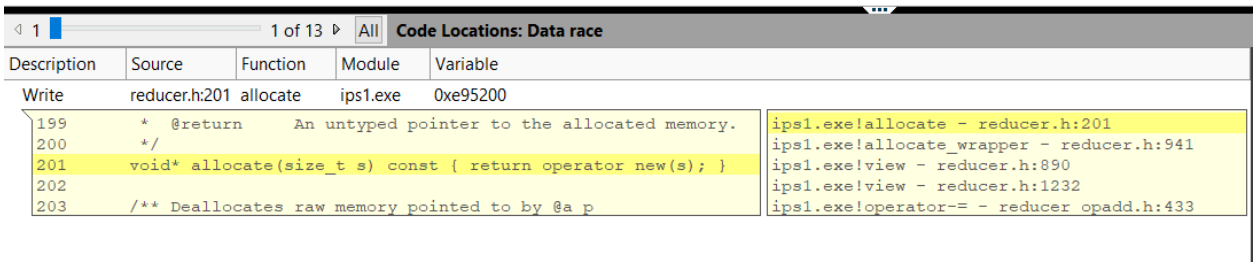


Была обнаружена зависимость по данным. Устраним ее:

```
void ParallelSerialGaussMethod(double **matrix, const int rows, double* result)
{
	int k;

	auto t0 = 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];
			}
		}
	}
	auto t1 = high_resolution_clock::now();

	// 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_k(matrix[k][rows]);

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

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

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

	duration<double> dur = t1 - t0;
	printf("Forward elimination time: %f\n", dur.count());
}
```

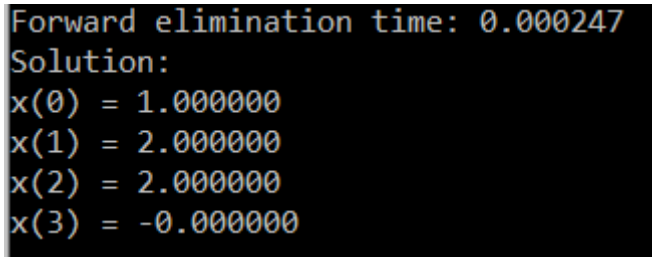
Отчет после исправления. Ошибка где-то в cilk.

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



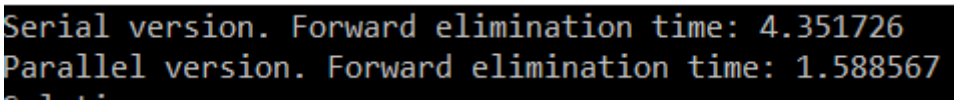
5. *Убедитесь на примере тестовой матрицы* **test_matrix** *в том, что функция, реализующая параллельный метод Гаусса работает правильно. Сравните время выполнения прямого хода метода Гаусса для последовательной и параллельной реализации при решении матрицы, имеющей количество строк* **MATRIX_SIZE,**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

Проверим на тестовой матрице:

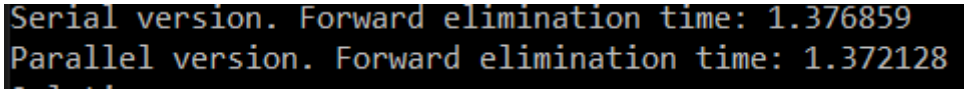


Решение правильное.

Запустим в Debug:



Запустим в Release с уровнем оптимизации -O2.



Видим, что в режиме Debug Есть ускорение, пример в 3 раза. Но в Release ускорения не наблюдается. Получается, что компилятор в данном случае может все соптимизировать. (Использовал VS2019).