

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:

```
Forward elimination time: 4.421223
Solution:
x(0) = 0.247359
x(1) = 3.873851
x(2) = -5.091191
x(3) = 0.227762
x(4) = -2.481059
x(5) = 4.129171
x(6) = 0.478180
x(7) = 3.240782
x(8) = 3.064479
x(9) = 1.810358
x(10) = 1.668213
x(11) = -0.477018
x(12) = 1.011570
x(13) = -5.034685
x(14) = -3.885078
x(15) = 3.307710
x(16) = 5.307995
x(17) = 1.249729
x(18) = 0.261155
x(19) = 0.689662
x(20) = 4.228978
x(21) = -0.901447
x(22) = 1.081886
x(23) = 1.085706
x(24) = 1.112112
x(25) = 2.996504
x(26) = 2.792385
x(27) = 0.891909
```

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.

Elapsed Time^②: 5.496s

➤ CPU Time^②: 4.422s

Total Thread Count: 1

Paused Time^②: 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 ^②
SerialGaussMethod	IPS1.exe	4.160s
rand	ucrtbased.dll	0.168s
InitMatrix	IPS1.exe	0.039s
_stdio_common_vfprintf	ucrtbased.dll	0.039s
malloc	ucrtbased.dll	0.016s

*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

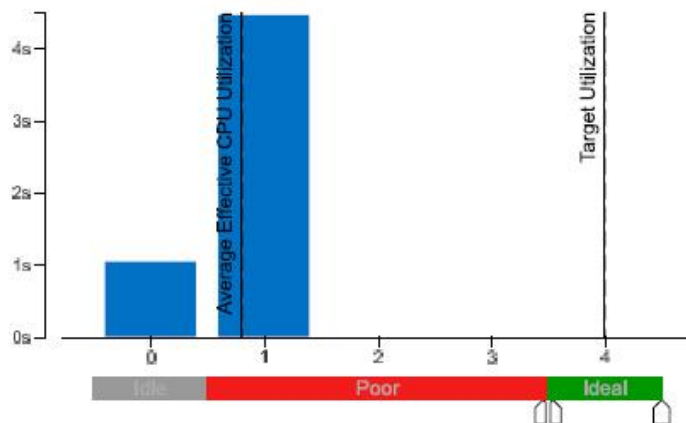
Parallelism^②: 20.1% 🚩

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.



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

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

    auto t0 = high_resolution_clock::now();
    // прямой ход метода Гаусса
    for (k = 0; k < rows; ++k)
    {
        //
        for (int i = k + 1; i < rows; ++i)
        {
            coef = -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

ID ▲	Type	Sources	Modules	State
P1	Data race	[Unknown]; IPS1.cpp	ips1.exe	Not fixed
P2	Data race	[Unknown]	ips1.exe	New
P3	Data race	[Unknown]	ips1.exe	Not fixed
P4	Data race	[Unknown]	ips1.exe	New
P5	Data race	[Unknown]	ips1.exe	Not fixed

Code Locations: Data race				
Description	Source	Function	Module	Variable
Read	ips1.exe:0x212b	[Unknown]	ips1.exe	block allocated at IPS1.cpp:146
Symbol information not found. Suggestion: Specify locations in a Project Properties dialog box search tab, then re-resolve the result.			ips1.exe!0x212b	
Write	IPS1.cpp:115	ParallelSerialGaussMethod	ips1.exe	block allocated at IPS1.cpp:146
<pre> 113 // 114 cilk_for (int j = k + 1; j < rows; ++j) 115 { 116 result[k] -= matrix[k][j] * result[j]; 117 } </pre>			ips1.exe!0x2142 ips1.exe!ParallelSerialGaussMethod - IPS1.cpp:	

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

Code Locations: Data race				
Description	Source	Function	Module	Variable
Write	reducer.h:201	allocate	ips1.exe	0xe95200
199	* @return An untyped pointer to the allocated memory.			
200	*/			
201	void* allocate(size_t s) const { return operator new(s); }			
202				
203	/** Deallocates raw memory pointed to by @a p			
			ips1.exe!allocate - reducer.h:201	
			ips1.exe!allocate_wrapper - reducer.h:941	
			ips1.exe!view - reducer.h:890	
			ips1.exe!view - reducer.h:1232	
			ips1.exe!operator-= - reducer_opadd.h:433	

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:

```
Forward elimination time: 0.000247
Solution:
x(0) = 1.000000
x(1) = 2.000000
x(2) = 2.000000
x(3) = -0.000000
```

Решение правильное.

Запустим в Debug:

```
Serial version. Forward elimination time: 4.351726
Parallel version. Forward elimination time: 1.588567
```

Запустим в Release с уровнем оптимизации -O2.

```
Serial version. Forward elimination time: 1.376859
Parallel version. Forward elimination time: 1.372128
```

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 оптимизировать. (Использовал VS2019).