1. **Запустите первоначальную версию программы и получите решение для тестовой матрицы test\_matrix, убедитесь в правильности приведенного алгоритма:**



Рис.1 – Результат выполнения программы с последовательным методом Гаусса

Исходная система:

Решение:

Следовательно, алгоритм является верным.

1. **Добавьте строки кода для измерения времени 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**).**

Результат:

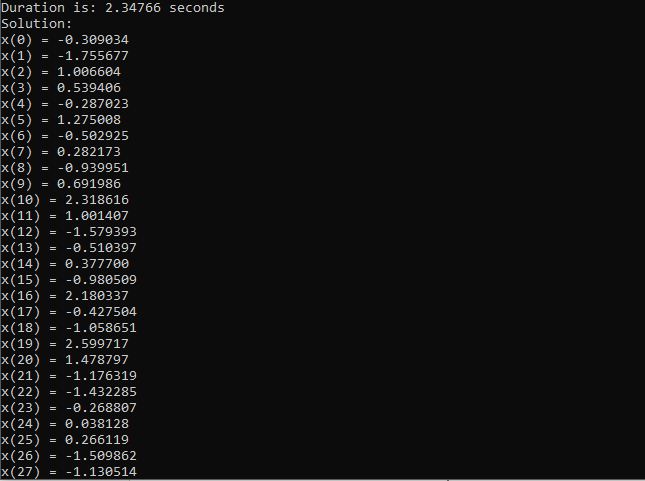


Рис.2 – Результат выполнения программы с последовательным методом Гаусса для матрицы размера MATRIX\_SIZE

В main были добавлены следующие строки кода:

double \*\*test\_matrix = new double\*[MATRIX\_SIZE];

for (i = 0; i < MATRIX\_SIZE; ++i)

{

test\_matrix[i] = new double[MATRIX\_SIZE + 1];

}

double \*result = new double[MATRIX\_SIZE];

InitMatrix(test\_matrix);

SerialGaussMethod(test\_matrix, MATRIX\_SIZE, result);

for (i = 0; i < MATRIX\_SIZE; ++i)

{

delete[]test\_matrix[i];

}

printf("Solution:\n");

for (i = 0; i < MATRIX\_SIZE; ++i)

{

printf("x(%d) = %lf\n", i, result[i]);

}

1. **С помощью инструмента** Amplifier XE **определите наиболее часто используемые участки кода новой версии программы. Сохраните скриншот результатов анализа**Amplifier XE***.***

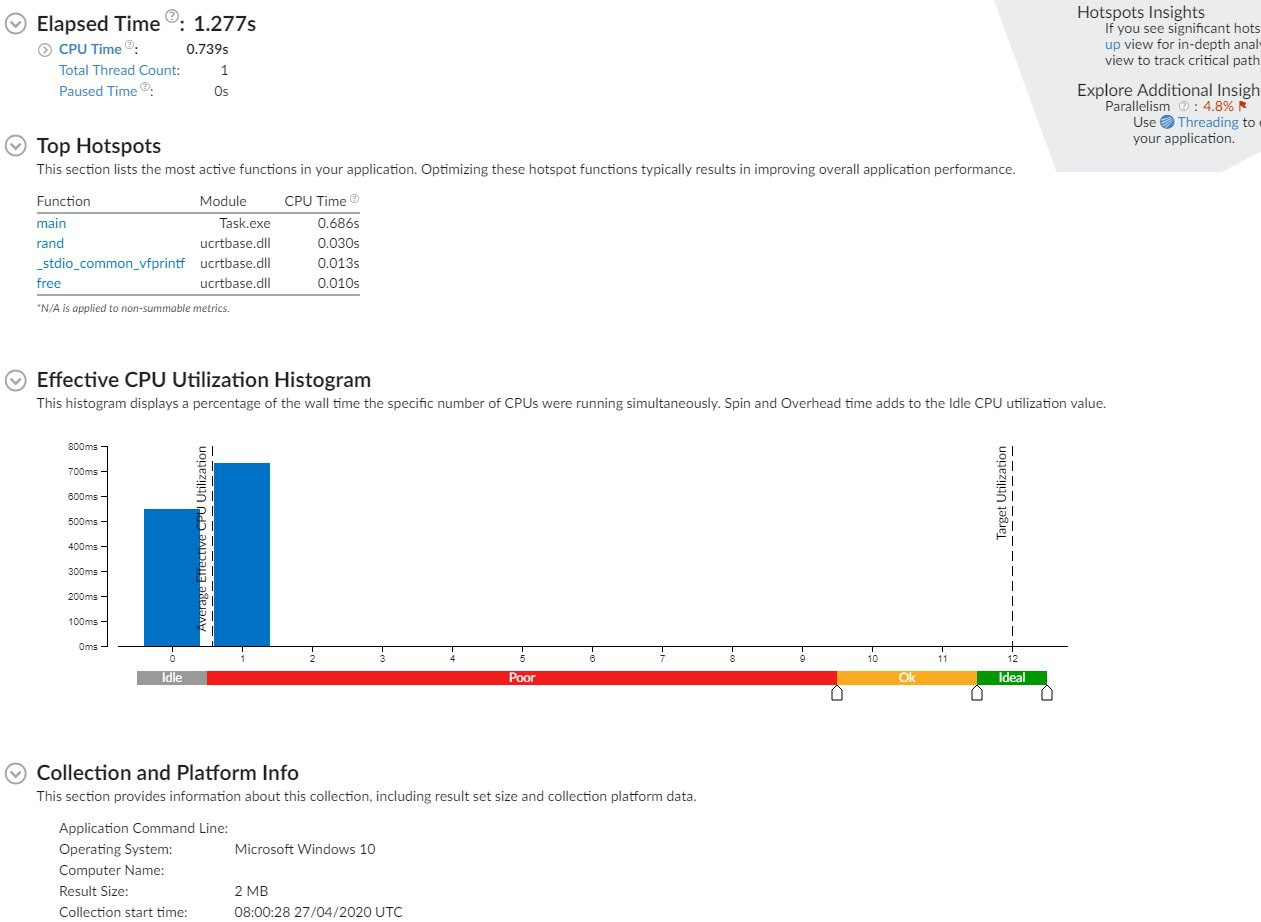


Рис.3 – Гистограмма и общее временя выполнения

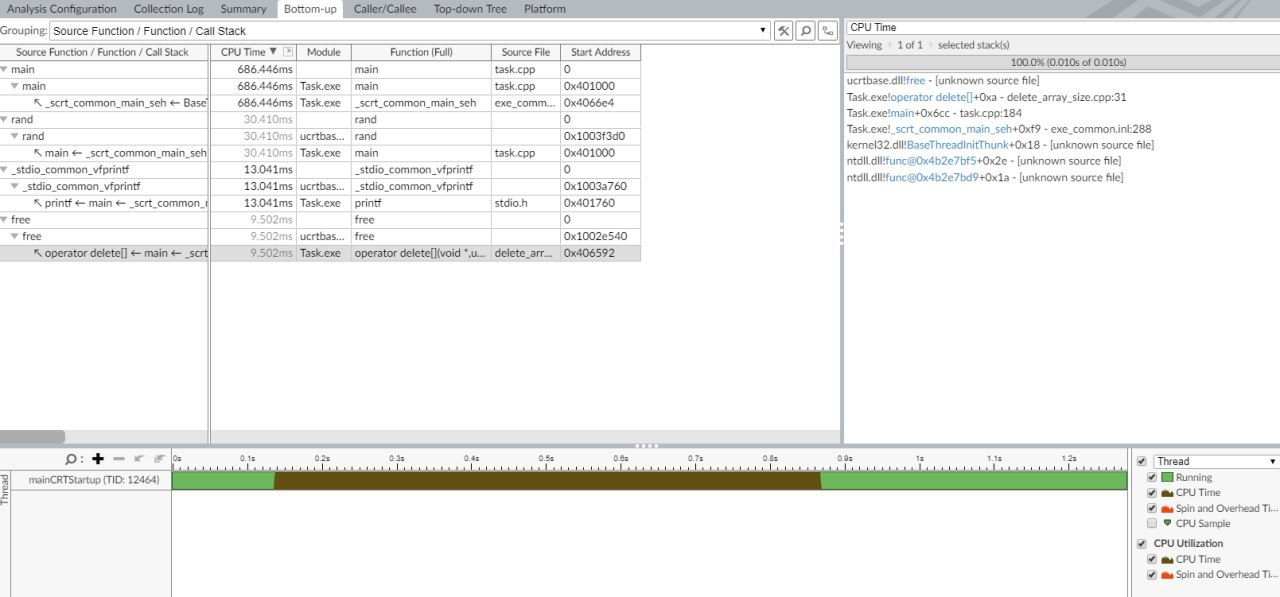


Рис.4 – Время выполнения каждой части программы

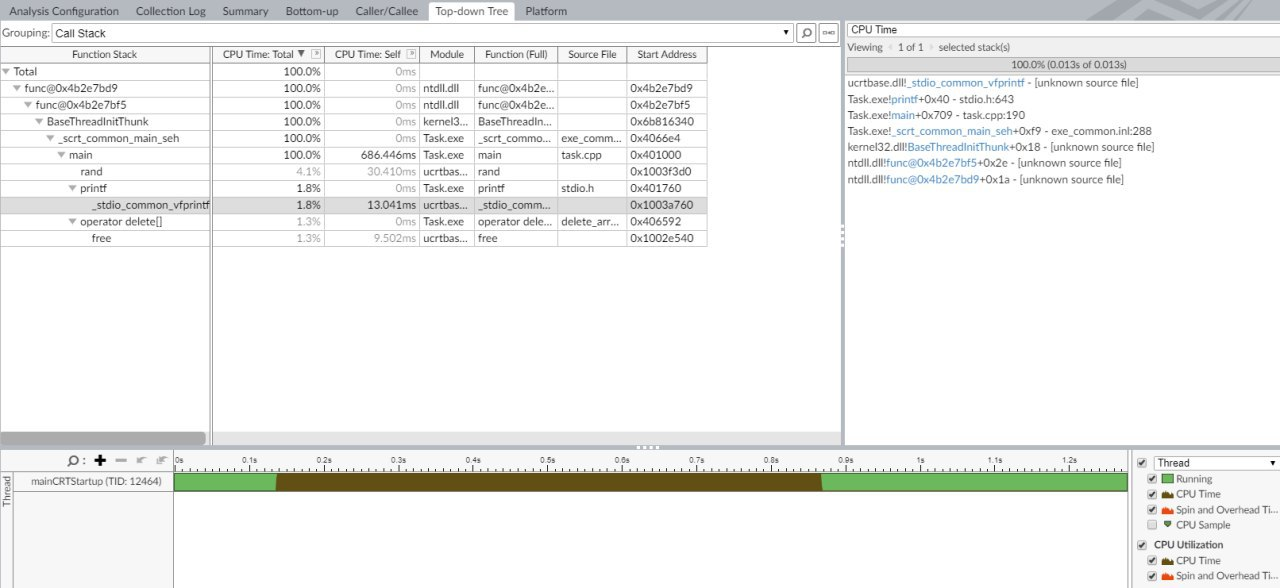


Рис.5 – Часть затраты времени процессора в процентах для функций

1. **Создайте, на основе последовательной функции***SerialGaussMethod()***, новую функцию, реализующую параллельный метод Гаусса. Введите параллелизм в новую функцию, используя** cilk\_for.

duration<double> 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) //здесь k - столбец (осн), i - строка

{

koef = -matrix[i][k] / matrix[k][k];

for (int j = k; j <= rows; ++j)//здесь k - строка, j - столбец

{

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

}

}

}

high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();

duration<double> duration = (t2 - t1);

std::cout << "Parallel Duration is: " << duration.count() << " seconds" << std::endl;

// 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< cilk::op\_add<double> > result\_p(matrix[k][rows]);

//

cilk\_for(int j = k + 1; j < rows; ++j)

{

\*result\_p += (-1)\*matrix[k][j] \* result[j];

}

result[k] = result\_p.get\_value();

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

}

return duration;

}

1. **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.**

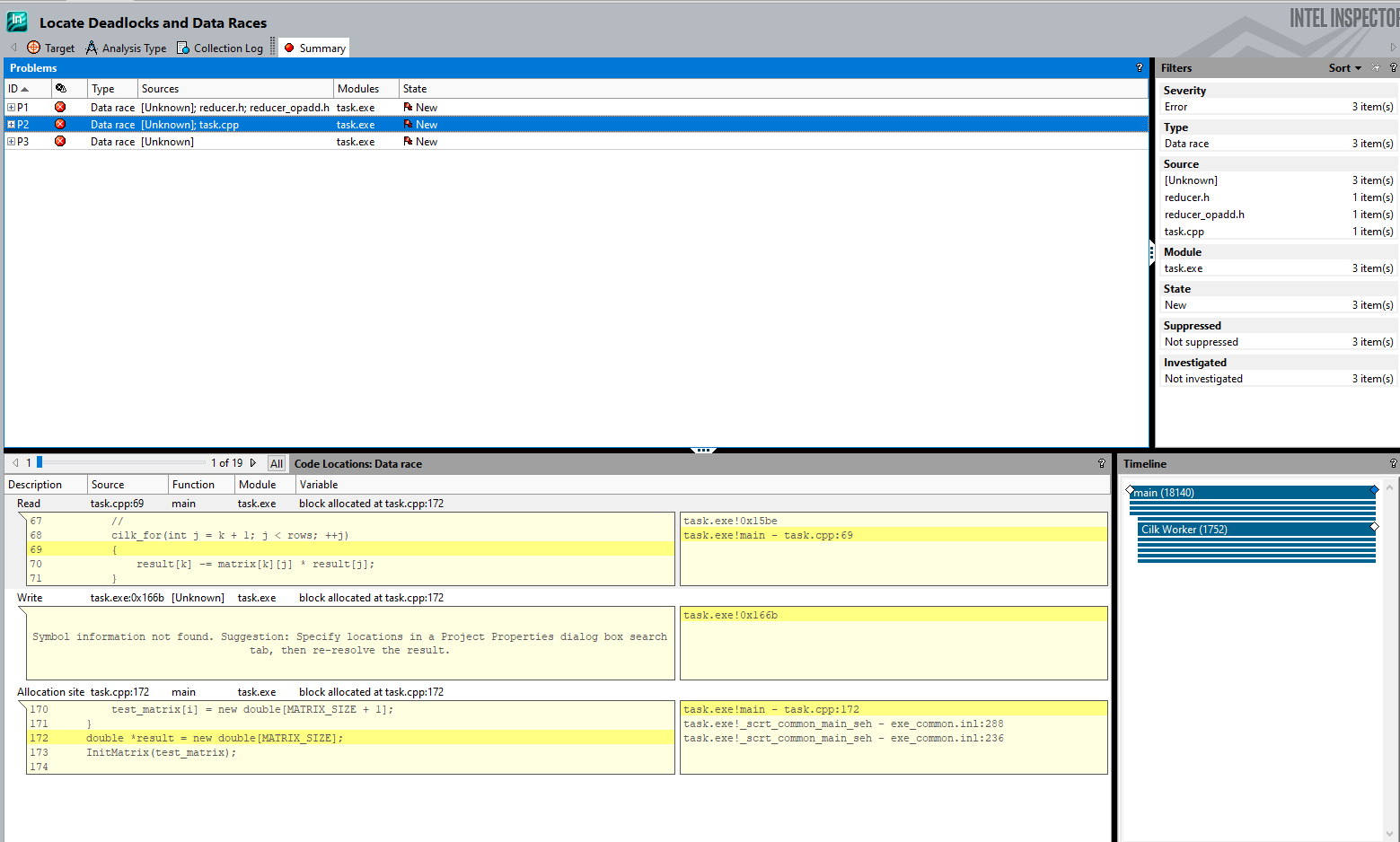


Рис.6 – Обнаружены гонки данных

Добавление в программу cilk\_reducer(в участок кода отвечающий за result, 105 строчка) и введение внутрь цикла общей переменной koef(строка 88) устранили гонки данных.

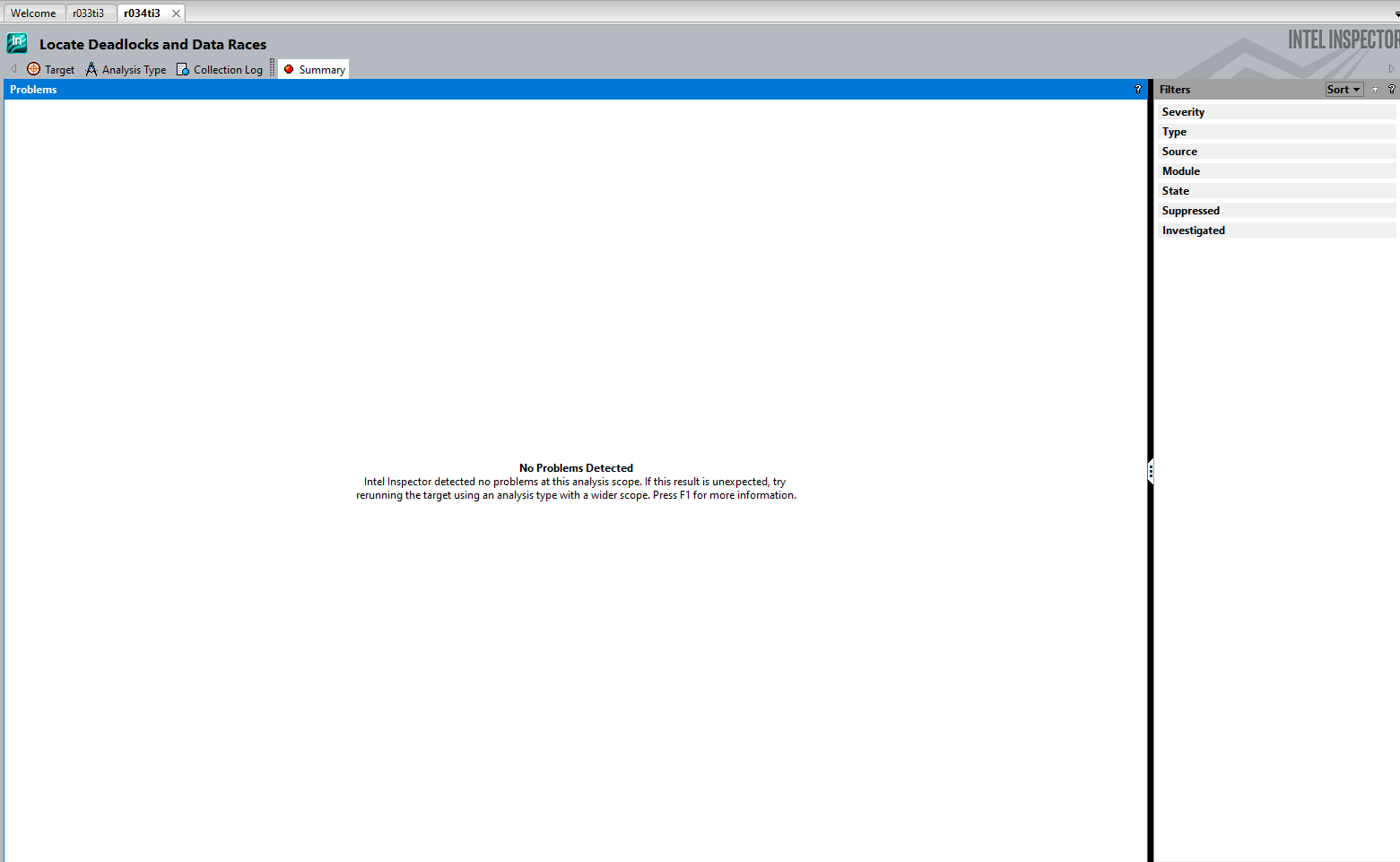


Рис.7 – Проблем не было обнаружено

1. **Убедитесь на примере тестовой матрицы**test\_matrix **в том, что функция, реализующая параллельный метод Гаусса работает правильно.**

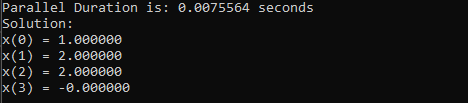


Рис.8 – Результат решения исходной матрицы параллельным методом Гаусса

Решение параллельным методом совпало с решением последовательным методом. Следовательно, функция *ParallelGaussMethod* работает верно.

1. **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.**

Ниже приведен листинг кода части функции main и результат выполнения.

double \*\*test\_matrix = new double\*[MATRIX\_SIZE];

for (i = 0; i < MATRIX\_SIZE; ++i)

{

test\_matrix[i] = new double[MATRIX\_SIZE + 1];

}

double \*result = new double[MATRIX\_SIZE];

InitMatrix(test\_matrix);

duration<double> parallel\_duration = ParallelGaussMethod(test\_matrix, MATRIX\_SIZE, result);

duration<double> serial\_duration = SerialGaussMethod(test\_matrix, MATRIX\_SIZE, result);

std::cout << "Result:" << std::endl;

std::cout << "Serial Duration is: " << serial\_duration.count() << " seconds" << std::endl;

std::cout << "Parallel Duration is: " << parallel\_duration.count() << " seconds" << std::endl;

std::cout << "Acceleration is: " << serial\_duration.count() - parallel\_duration.count() << " seconds" << std::endl;

for (i = 0; i < MATRIX\_SIZE; ++i)

{

delete[]test\_matrix[i];

}

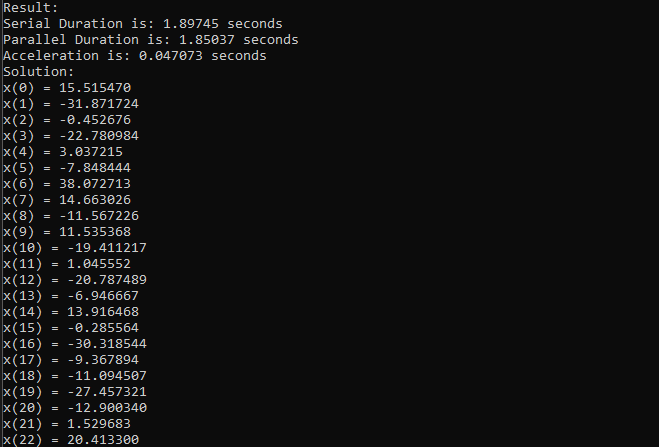


Рис.9 – Время выполнения последовательного и параллельного метода Гаусса и полученное ускорение для матрицы размера MATRIX\_SIZE

**Выводы:** в рамках данной лабораторной работы мы познакомились с инструментами Vtune Amplifier и Inspector.

При выполнении работы использовалась Visual Studio 2017 и IPS 2019.

Операционная система: Windows 10

Процессор: 2.80ГГц, 4 физических ядра, 8 логических ядер