

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра вычислительной техники

ОТЧЕТ
по лабораторной работе № 1
по дисциплине «Параллельные алгоритмы и системы»
Тема: Оптимизация программы

Студент гр 9306

Преподаватель

Евдокимов О.В.

Пазников А. А.

Санкт-Петербург
2023

Оглавление

Цель работы.....	3
Задание.....	3
Ход работы.....	3
Описание алгоритма.....	3
Демонстрация корректности работы алгоритма.....	4
Выбор компилятора.....	4
Код программы.....	5
Многопоточная реализация с OpenMP.....	7
Код программы.....	10
Оптимизация с помощью векторных инструкций.....	12
Код программы.....	14
Выводы.....	17

Цель работы

Изучение методов оптимизации программ для более полного использования аппаратных мощностей системы.

Задание

Оптимизировать программу выполняющую решение системы линейных алгебраических уравнений (СЛАУ) методом Гаусса.

Ход работы

Программа получает от пользователя расширенную матрицу системы и выдает вектор ответ, содержащий значения неизвестных.

Описание алгоритма

Алгоритм состоит из двух частей:

- приведение матрицы к треугольному виду;
- обратная подстановка для вычисления значения неизвестных.

Рассмотрим сначала первую часть. В цикле по количеству неизвестных - 1 мы выполняем вычитание из каждой строки, ниже строки с неизвестной на главной диагонали, строку с неизвестной умноженной на отношение: текущий элемент под переменной на главной диагонали к элементу на главной диагонали. Таким образом с каждой итерацией внешнего цикла мы зануляем все элементы под главной диагональю.

Псевдокод, где a_{ij} - элемент на i строке j столбце, n – количество переменных:

for i in $[1:n-1]$:

for j in $[i+1:n]$:

$$a_{j_} = a_{j_} - \frac{a_{ji}}{a_{ii}} a_{i_}$$

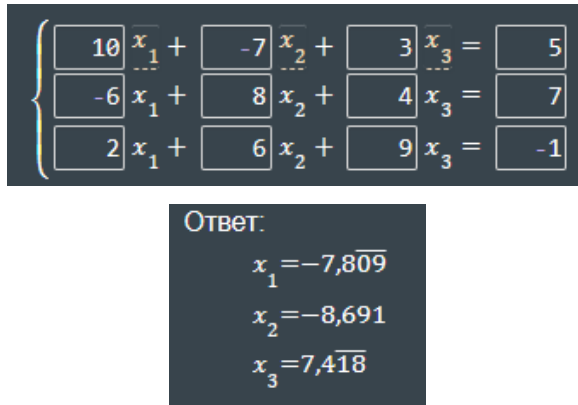
Поскольку вычитание строк матрицы также выполняется за n шагов. Таким образом получаем, что ассимптотическая сложность данной части $O(n^3)$.

При обратной подстановке мы сначала последней неизвестной присваиваем значение полученное как: правая часть равенства (последний элемент приставленного вектора) деленная на левую часть равенства (последний элемент на главной диагонали). Затем поднимаясь вверх по неравенствам в цикле вычитаем из правой части неизвестные умноженные на соответствующие коэффициенты из строки. Затем делением на элемент на главной диагонали получаем неизвестное. Таким образом ассимптотическая сложность

данной части $O(n^2)$.

Демонстрация корректности работы алгоритма

На рисунке 1 приведен результат решения СЛАУ с помощью сайта (matrixcalc.org).



The image shows a screenshot of a website interface for solving a system of linear equations. It displays three equations in a set of braces, with coefficients and variables in boxes. Below the equations, it shows the solution for x1, x2, and x3.

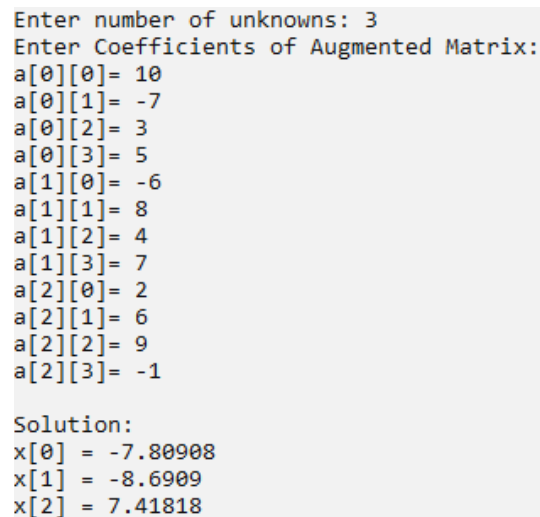
$$\begin{cases} 10x_1 + (-7)x_2 + 3x_3 = 5 \\ -6x_1 + 8x_2 + 4x_3 = 7 \\ 2x_1 + 6x_2 + 9x_3 = -1 \end{cases}$$

Ответ:

$$\begin{aligned} x_1 &= -7,809 \\ x_2 &= -8,691 \\ x_3 &= 7,418 \end{aligned}$$

Рисунок 1 — решение системы при помощи сайта.

На рисунке 2 приведен результат решения СЛАУ написанной нами программы. Как мы видим, результаты совпали, значит можно подозревать корректность написанной программы.



The image shows a screenshot of a program's output. It prompts the user to enter the number of unknowns (3) and then the coefficients of the augmented matrix. It then displays the solution for x[0], x[1], and x[2].

```
Enter number of unknowns: 3
Enter Coefficients of Augmented Matrix:
a[0][0]= 10
a[0][1]= -7
a[0][2]= 3
a[0][3]= 5
a[1][0]= -6
a[1][1]= 8
a[1][2]= 4
a[1][3]= 7
a[2][0]= 2
a[2][1]= 6
a[2][2]= 9
a[2][3]= -1

Solution:
x[0] = -7.80908
x[1] = -8.6909
x[2] = 7.41818
```

Рисунок 2 — решение системы при помощи программы.

Выбор компилятора

Проведем тестирование и анализ влияния различных компиляторов на скорость работы программы. Имеем глобальную расширенную матрицу размера 4096 на 4096. Заполним данную матрицу случайными числами не до конца, а только область 1024 на 1024 (вся матрица считается слишком долго). Запустим решение данной матрицы 128 раз с количеством неизвестных равным 1024 и замерим время каждого решения. Будем использовать три различных компилятора GCC (Msys64 MinGW64 GNU 12.2.0), Clang (Visual Studio clang_cl_x64_x64 Clang 15.0.1), MSVC (MSVC 19.35.32216.1) с ключами

оптимизации -O0. На рисунке 3 приведен график ящиков с усами времени с удаленными выбросами (после этого действия в наборе данных осталось 99 записей). Также на рисунке 4 приведена столбчатая диаграмма математического ожидания для каждого компилятора со стандартным отклонением.

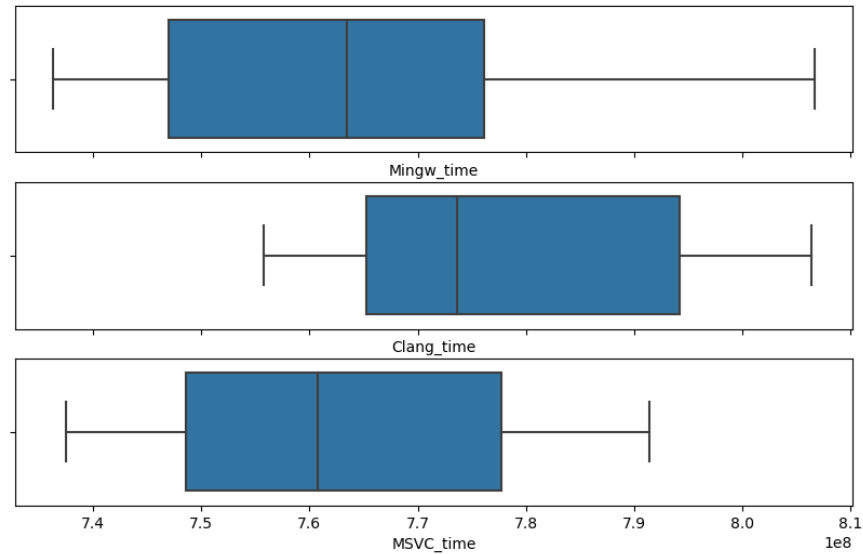


Рисунок 3 — ящик с усами компиляторов базовой реализации.

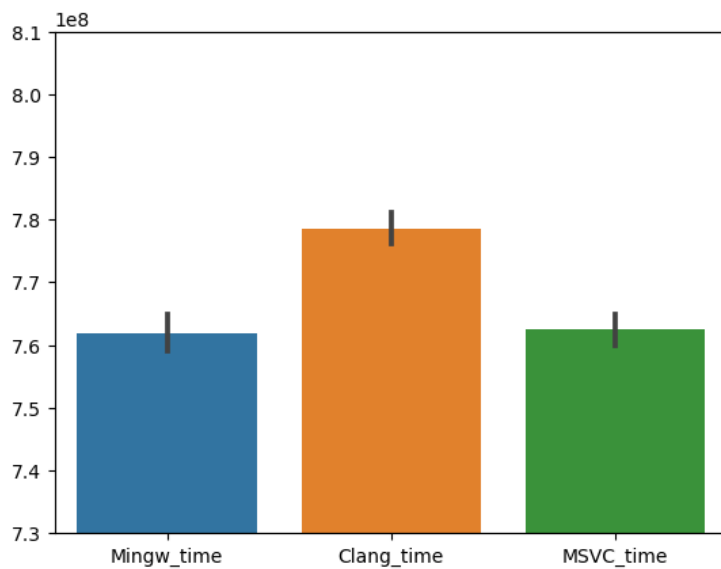


Рисунок 4 — столбчатая диаграмма компиляторов базовой реализации.

Как мы видим по данным рисункам лучшее время показывает MSVC, худшее Clang.

Код программы

Приведем ниже код программы выполняющей ранее описанные действия. Результаты измерения времени каждой итерации вычислений выводятся в .csv файл. Данная программа сразу написана так, чтобы в дальнейшем мы могли проверять на корректность новые реализации (x_res) сравнивая ответ с данной реализацией (x_base от calc_base)

```

constexpr int SIZE = 4096;
constexpr int max_iter = 128;

float a_res[SIZE][SIZE], a_base[SIZE][SIZE], x_res[SIZE], x_base[SIZE];
int n = 1024;

void calc_base(float a[SIZE][SIZE], float x[SIZE])
{
    /* Applying Gauss Elimination */
    for (int i = 0; i < n - 1; ++i)
    {
        if (a[i][i] == 0.0)
        {
            throw std::runtime_error("Mtx is singular");
        }
        for (int j = i + 1; j < n; ++j)
        {
            const float ratio = a[j][i] / a[i][i];

            for (int k = i; k < n + 1; ++k)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }
    /* Obtaining Solution by Back Substitution Method */
    x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

    for (int i = n - 2; i >= 0; --i)
    {
        float sum = 0;
        for (int j = i + 1; j < n; ++j)
        {
            sum += a[i][j] * x[j];
        }
        x[i] = (a[i][n] - sum) / a[i][i];
    }
}

bool is_eq(float l[SIZE], float r[SIZE])
{
    float epsilon = 0.0001;
    bool res = true;
    for (int i = 0; i < n; ++i)
    {
        float d = fabs(l[i] - r[i]);
        if (!(d < epsilon))
            res = false;
    }

    return res;
}

int main()
{
    srand(std::chrono::system_clock::now().time_since_epoch().count());

    std::fstream file;
    file.open("TimeMeasure.csv", std::ios_base::out | std::ios_base::trunc);
    file << "time\n";

    for (int i = 0; i < n; ++i)
    {

```

```

        for (int j = 0; j < n + 1; ++j)
        {
            float r = rand();
            a_base[i][j] = r;
            a_res[i][j]=r;
        }
    }

    //calc_base(a_base, x_base);
    //calc_omp(a_res, x_res);
    //
    //if (!is_eq(x_base, x_res))
    //    throw std::runtime_error("Test not pass");

    for (int cur_iter = 0; cur_iter < max_iter; ++cur_iter)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n + 1; ++j)
            {
                float r = rand();
                a_res[i][j] = r;
            }
        }

        std::cout << cur_iter << std::endl;

        std::chrono::time_point<std::chrono::high_resolution_clock> t1 =
std::chrono::high_resolution_clock::now();

        calc_base(a_res, x_res);

        auto elapsed_time = std::chrono::high_resolution_clock::now() -
t1;
        file << elapsed_time.count() << "\n";
    }

    file.close();

    return 0;
}

```

Многопоточная реализация с OpenMP

Дополним нашу программу распараллеливанием, чтобы использовать имеющиеся на устройстве ядра процессора. Для этого будем использовать достаточно популярную библиотеку OpenMP.

Первое изменение которое мы можем внести — это распараллеливание вычитания верхней строки из нижних строк при составлении треугольной матрицы. Этот процесс не имеет никакой зависимости по данным и может быть реализован лишь одной строчкой `#pragma omp parallel for` до объявления цикла.

Также можно дополнительно распараллелить само вычитание двух строк. Однако не факт, что это даст нам хорошее ускорение, потому что будет слишком большое количество созданий потоков. Давайте проведем эксперимент. Эксперимент будем проводить для трех

ранее указанных компиляторов с их версиями OpenMp: Clang (5.0), MinGW (4.5), MSVC (2.0). Без явного указания количества потоков (автоматический выбор OpenMP) получили результаты представленные на рисунках 5 и 6. Процессор: Intel Core i7-8700 с 6 ядрами и 12 потоками. По данным рисункам мы можем сделать вывод о примерно 4х кратном ускорении.

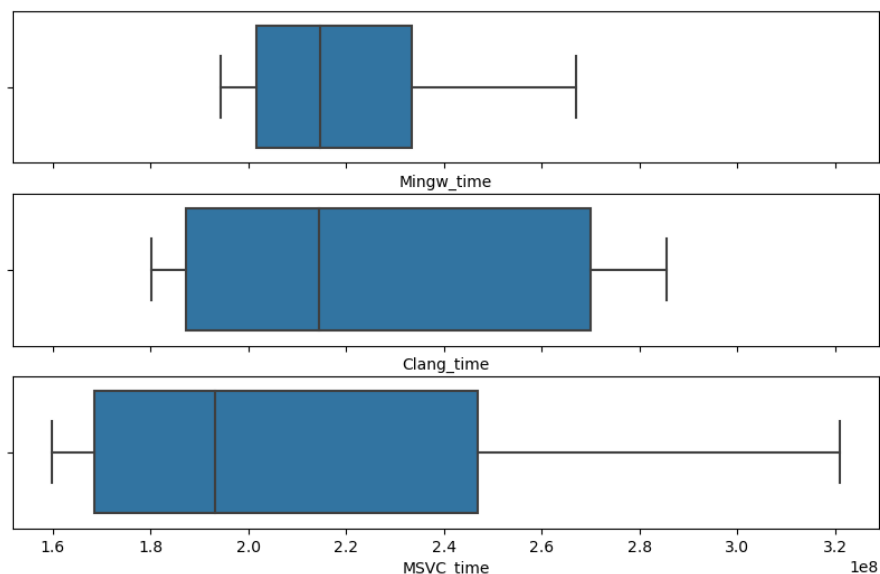


Рисунок 5 — ящик с усами компиляторов первой многопоточной реализации.

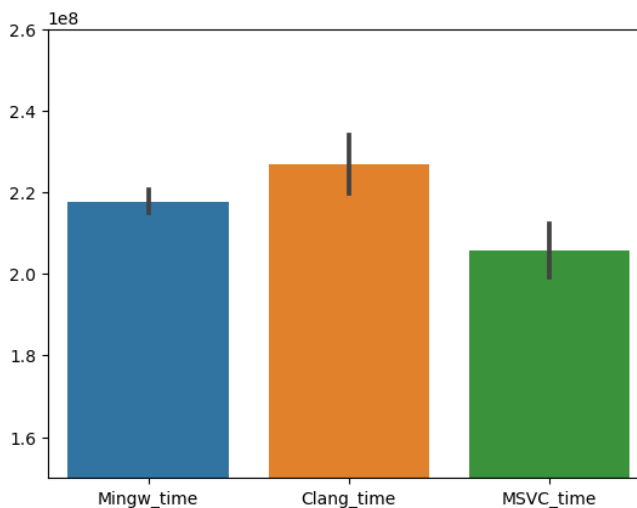


Рисунок 6 — столбчатая диаграмма компиляторов первой многопоточной реализации.

Теперь проведем эксперимент добавив `#pragma omp parallel for` для вложенного цикла, вычитания двух строк. Результаты эксперимента приведены на рисунке 7. Как мы видим такая реализация ухудшила результаты при использовании всех компиляторов, больше всего результаты MinGW.

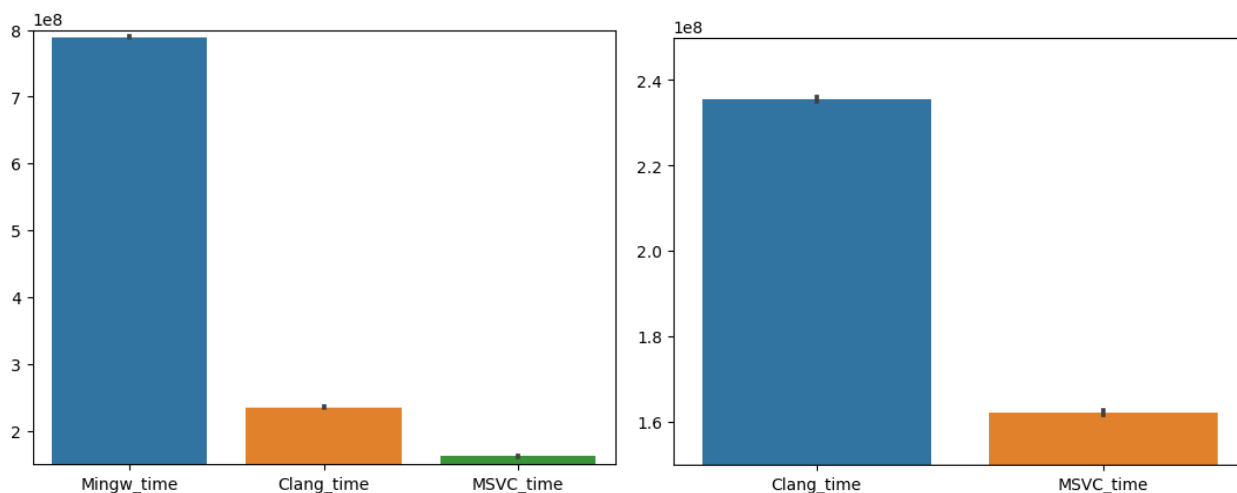


Рисунок 7 — столбчатая диаграмма компиляторов второй многопоточной реализации.

Давайте перейдем к прошлой реализации и поэкспериментируем с количеством потоков. В качестве компилятора выберем MSVC, как показывающий наилучший результат. На рисунке 8 показан график изменения времени вычисления (слева) в зависимости от количества явно указанных потоков для блока `parallel for` и относительный прирост, справа. Как мы видим по данным графикам, наилучшая производительность достигается при использовании 12 потоков, что логично, поскольку система на, которой запускается программа, тоже имеет 12 потоков процессора.

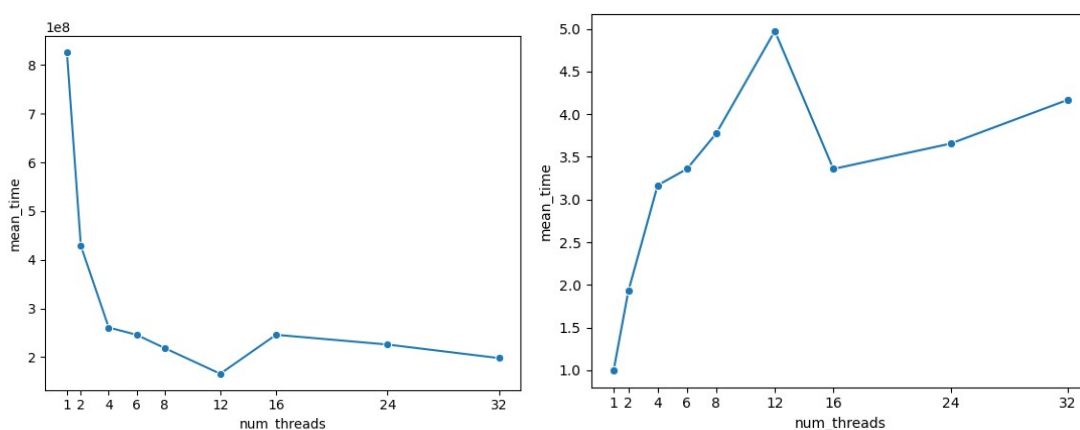


Рисунок 8 — график эффективности относительно количества потоков.

Дальше к данной реализации мы можем добавить распараллеливание процесса обратной подстановки. В данном процессе мы можем распараллелить только внутренний цикл аккумулярования левой части неравенства, для последующего вычитания из правой части. Для этого добавим директиву `#pragma omp parallel`. Количество работы, которое нам нужно выполнить равно (количество переменных - номер текущего неравенства). Эту работу мы разделяем на `nthreads`, затем назначаем им верхние и нижние границы элементов строки, которые каждый должен пройти, причем последний всегда проходит до конца строки для

обработки случаев, когда количество работы не кратно количеству потоков. Каждый поток аккумулирует эту сумму в приватную переменную `pr_sum` затем по завершении своего цикла, входит в критическую секцию и прибавляет к общей переменной суммы (`sh_sum`) свою приватную, таким образом мы исключаем гонки данных.

Проведем измерение времени выполнения. Результаты измерений показаны на рисунке 9. По данным результатам мы видим, что изменение практически не повлияло на время выполнения, можно даже сказать о деградации времени выполнения. Кроме того пришлось увечить допуск на равенство чисел с плавающей точкой, при сравнении результатов от последовательной и данной реализации, до одной десятой. Причина деградации, скорее всего, в процессе синхронизации.

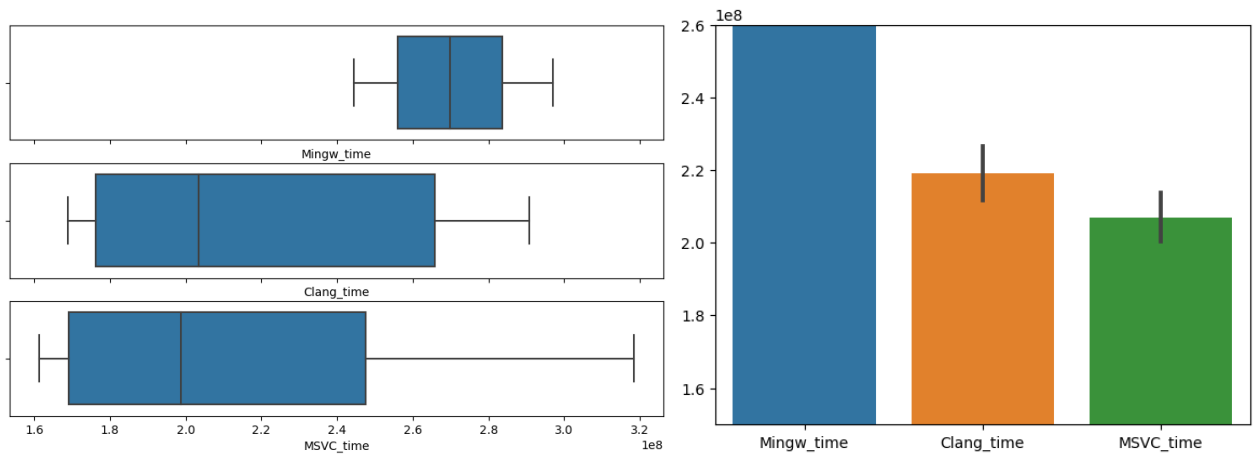


Рисунок 9 — результаты времени выполнения третьей многопоточной реализации.

Код программы

```
constexpr int SIZE = 4096;
constexpr int max_iter = 128;

__declspec(align(16)) float a_res[SIZE][SIZE], a_base[SIZE][SIZE], x_res[SIZE],
x_base[SIZE];
int n = 1024;

void calc_base(float a[SIZE][SIZE], float x[SIZE])
{
    /* Applying Gauss Elimination */
    for (int i = 0; i < n - 1; ++i)
    {
        if (a[i][i] == 0.0)
        {
            throw std::runtime_error("Mtx is singular");
        }
        for (int j = i + 1; j < n; ++j)
        {
            const float ratio = a[j][i] / a[i][i];

            for (int k = i; k < n; ++k)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }
}
```

```

    }
}
/* Obtaining Solution by Back Substitution Method */
x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)
{
    float sum = 0;
    for (int j = i + 1; j < n; ++j)
    {
        sum += a[i][j] * x[j];
    }
    x[i] = (a[i][n] - sum) / a[i][i];
}

}

void calc_omp(float a[SIZE][SIZE], float x[SIZE])
{
    /* Applying Gauss Elimination */
    for (int i = 0; i < n - 1; ++i)
    {
        if (a[i][i] == 0.0)
        {
            throw std::runtime_error("Mtx is singular");
        }

#pragma omp parallel for num_threads(12)
        for (int j = i + 1; j < n; ++j)
        {
            const float ratio = a[j][i] / a[i][i];
            for (int k = i; k < n + 1; ++k)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }

    /* Obtaining Solution by Back Substitution Method */
    x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

    for (int i = n - 2; i >= 0; --i)
    {
        float sum = 0;
        for (int j = i + 1; j < n; ++j)
        {
            sum += a[i][j] * x[j];
        }
        x[i] = (a[i][n] - sum) / a[i][i];
    }
}

bool is_eq(float l[SIZE], float r[SIZE])
{
    float epsilon = 0.1;
    bool res = true;
    for (int i = 0; i < n; ++i)
    {
        float d = fabs(l[i] - r[i]);

        if (!(d < epsilon))
            res = false;
    }

    return res;
}

```

```

int main()
{
    srand(std::chrono::system_clock::now().time_since_epoch().count());

    std::fstream file;
    file.open("TimeMeasure.csv", std::ios_base::out | std::ios_base::trunc);
    file << "time\n";

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n + 1; ++j)
        {
            float r = rand();
            a_base[i][j] = r;
            a_res[i][j] = r;
        }
    }

    calc_base(a_base, x_base);
    calc_omp(a_res, x_res);

    if (!is_eq(x_base, x_res))
    {
        std::cout << "Test not pass\n";
        throw std::runtime_error("Test not pass");
    }

    for (int cur_iter = 0; cur_iter < max_iter; ++cur_iter)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n + 1; ++j)
            {
                float r = rand();
                a_res[i][j] = r;
            }
        }

        std::cout << cur_iter << std::endl;

        std::chrono::time_point<std::chrono::high_resolution_clock> t1 =
std::chrono::high_resolution_clock::now();

        calc_omp(a_res, x_res);

        auto elapsed_time = std::chrono::high_resolution_clock::now() -
t1;
        file << elapsed_time.count() << "\n";
    }

    file.close();
    return 0;
}

```

Оптимизация с помощью векторных инструкций

Мы пытались до этого добавить параллельное выполнение вложенного цикла приведения матрицы к треугольному виду. Однако добавление много поточности в данном случае не дало желаемого результата. Поэтому следующее, что мы можем сделать это

использовать векторные (SIMD) инструкции процессора для вычитания двух строк матрицы.

Первое, что нам нужно изменить это выровнять начало нашего массива по 16 байт, причем для сохранения выравненности строк, количество элементов в строке также должно быть кратно 4 (поскольку один float 4 байта).

Далее мы загружаем отношение в векторный регистр и в цикле от номера группы в которую попала текущая переменная до округленного вверх количества групп выполняем умножение, вычитание и запись с помощью соответствующих векторных инструкций. Так после измерения были получены графики времени выполнения программы приведенные на рисунке 10. По данным и прошлым графикам делаем вывод о примерно 70% приросте производительности

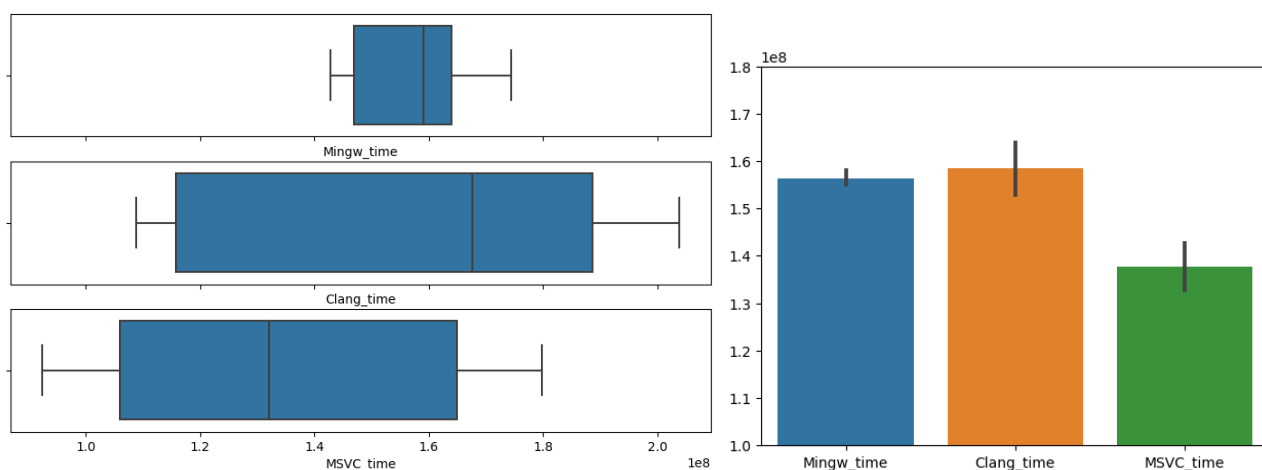


Рисунок 10 — графики времени выполнения первой части векторизации.

Далее векторизуем часть программы с вычислением значения переменных. Для это в теле первого цикла по неизвестным проверяем выравнен ли элемент с которого нужно анчать суммирование по сетке из количества бит на векторный регистр. Если нет, то считаем сумму обычным циклом, пока не выровняем индекс текущего элемента. Если индекс выровнен мы можем проверить имеются ли вообще какие-то группы по количеству элементов в векторном регистре, если да считаем сумму с использованием соответствующих векторных инструкций. Если количество неизвестных не было кратно количеству элементов в регистре, то нам нужно еще пройти циклом до конца строки матрицы. Были проведены измерения времени выполнения данной программы графики приведены на рисунке 11. По данным и прошлым графикам мы видим 60% улучшение производительности.

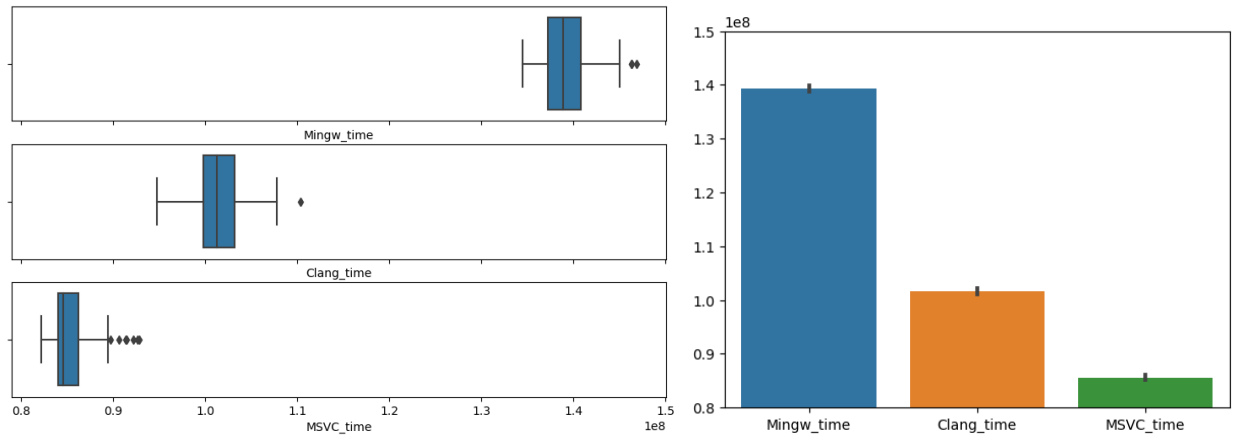


Рисунок 11 — графики времени выполнения второй части векторизации.

Код программы

```
constexpr int SIZE = 4096;
constexpr int max_iter = 128;

__declspec(align(16)) float a_res[SIZE][SIZE], a_base[SIZE][SIZE], x_res[SIZE],
x_base[SIZE];
int n = 1024;

void calc_base(float a[SIZE][SIZE], float x[SIZE])
{
    /* Applying Gauss Elimination */
    for (int i = 0; i < n - 1; ++i)
    {
        if (a[i][i] == 0.0)
        {
            throw std::runtime_error("Mtx is singular");
        }
        for (int j = i + 1; j < n; ++j)
        {
            const float ratio = a[j][i] / a[i][i];

            for (int k = i; k < n + 1; ++k)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }
    /* Obtaining Solution by Back Substitution Method */
    x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

    for (int i = n - 2; i >= 0; --i)
    {
        float sum = 0;
        for (int j = i + 1; j < n; ++j)
        {
            sum += a[i][j] * x[j];
        }
        x[i] = (a[i][n] - sum) / a[i][i];
    }
}

void calc_omp_vec128(float a[SIZE][SIZE], float x[SIZE])
{
    const int count_of_float_in_reg = sizeof(__m128) / sizeof(float);
```

```

/* Applying Gauss Elimination */
for (int i = 0; i < n - 1; ++i)
{
    if (a[i][i] == 0.0)
    {
        throw std::runtime_error("Mtx is singular");
    }
#pragma omp parallel for num_threads(12)
    for (int j = i + 1; j < n; ++j)
    {
        const __m128 m_ratio = _mm_set1_ps(a[j][i] / a[i][i]);
        int end = (n + 1) / count_of_float_in_reg +
static_cast<int>((n + 1) % count_of_float_in_reg != 0);
        for (int g = i / count_of_float_in_reg; g < end; ++g)
        {
            //a[j][k] = a[j][k] - ratio * a[i][k];

            __m128* row_j =
reinterpret_cast<__m128*>(&a[j][g * count_of_float_in_reg]);
            __m128* row_i =
reinterpret_cast<__m128*>(&a[i][g * count_of_float_in_reg]);

            __m128 r_i_mul = _mm_mul_ps(m_ratio, *row_i);
            __m128 sub = _mm_sub_ps(*row_j, r_i_mul);

            _mm_store_ps(&a[j][g *
count_of_float_in_reg], sub);
        }
    }
}
/* Obtaining Solution by Back Substitution Method */
x[n - 1] = a[n - 1][n] / a[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)
{
    float sum = 0;
    __m128 vsum = _mm_set1_ps(0.0f);
    int g = i + 1;
    int rem = (i + 1) % count_of_float_in_reg;
    if (rem != 0)
    {
        for (int k = rem, g = i + 1; k < count_of_float_in_reg;
++k, ++g)
        {
            sum += a[i][g] * x[g];
        }
    }
    // g is aligned to count_of_float_in_reg
    int end = n / count_of_float_in_reg;
    g = g / count_of_float_in_reg;
    if (end - g > 0)
    {
        for (; g < end; ++g)
        {
            __m128* a_i = reinterpret_cast<__m128*>(&a[i]
[g * count_of_float_in_reg]);
            __m128* x_g = reinterpret_cast<__m128*>(&x[g
* count_of_float_in_reg]);

            __m128 mul = _mm_mul_ps(*a_i, *x_g);
            vsum = _mm_add_ps(vsum, mul);
        }
        vsum = _mm_hadd_ps(vsum, vsum);
    }
}

```

```

        vsum = _mm_hadd_ps(vsum, vsum);
        _mm_store_ss(&sum, vsum);
    }

    for (int j = g * count_of_float_in_reg; j < n; ++j)
    {
        sum += a[i][j] * x[j];
    }

    x[i] = (a[i][n] - sum) / a[i][i];
}

}

bool is_eq(float l[SIZE], float r[SIZE])
{
    float epsilon = 0.1;
    bool res = true;
    for (int i = 0; i < n; ++i)
    {
        float d = fabs(l[i] - r[i]);

        if (!(d < epsilon))
            res = false;
    }

    return res;
}

int main()
{
    srand(std::chrono::system_clock::now().time_since_epoch().count());

    std::fstream file;
    file.open("TimeMeasure.csv", std::ios_base::out | std::ios_base::trunc);
    file << "time\n";

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n + 1; ++j)
        {
            float r = rand();
            a_base[i][j] = r;
            a_res[i][j] = r;
        }
    }

    calc_base(a_base, x_base);
    calc_omp_vec128(a_res, x_res);

    if (!is_eq(x_base, x_res))
    {
        std::cout << "Test not pass\n";
        throw std::runtime_error("Test not pass");
    }

    for (int cur_iter = 0; cur_iter < max_iter; ++cur_iter)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n + 1; ++j)
            {
                float r = rand();

```



```

        a_res[i][j] = r;
    }
}

std::cout << cur_iter << std::endl;

std::chrono::time_point<std::chrono::high_resolution_clock> t1 =
std::chrono::high_resolution_clock::now();

calc_omp_vec128(a_res, x_res);

auto elapsed_time = std::chrono::high_resolution_clock::now() -
t1;
    file << elapsed_time.count() << "\n";
}

file.close();

return 0;
}

```

Выводы

В ходе выполнения лабораторной работы мы изучили и применили на практике два метода оптимизации, а именно: многопоточная реализация и векторизация. Сравнивали данные реализации по математическому ожиданию времени выполнения, сводная таблица результатов приведена в таблице 1 с приростом относительно базовой реализации. Кроме сравнения времени выполнения реализации мы также изучили влияние компилятора на производительность программы. В итоге оказалось, что лучший результат, на операционной системе windows 10, показывает MSVC. Также узнали что лучшая производительность достигается при использовании 12 потоков.

Таблица 1

Реализация	Компилятор	Время выполнения	Прирост
Базовая	MinGW	7.63	1
	Clang	7.73	1
	MSVC	7.61	1
Многопоточная лучшая	MinGW	2.17	3.51
	Clang	2.25	3.43
	MSVC	2.1	3.62
Векторизованная лучшая	MinGW	1.4	5,45
	Clang	1	7.73
	MSVC	0.7	10.8