Государственное образовательное учреждение высшего профессионального образования
«Московский Государственный Технический Университет имени Н. Э. Баумана»

ОТЧЕТ
По лабораторной работе №4
По курсу «Анализ алгоритмов»
Тема: **«Многопоточное умножение матриц»**

Студент: Кононенко С. Д.
Группа: ИУ7-51

Москва, 2017

Постановка задачи

1. Реализовать алгоритм умножения матриц посредством N потоков
2. Сравнить алгоритм многопоточного умножения матриц с однопоточным
3. Провести эксперименты с замерами времени на матрицах разных размерностей
4. Сделать выводы об эффективности многопоточного умнодния матриц

Однопоточное умножение матриц

```
matrix_t matr_mult(const matrix_t m1, const matrix_t m2, unsigned long long
int *t)
{
    matrix_t res = create_matrix(m1.n, m2.m);
    *t = tick();
    for (int i = 0; i < m1.n; i++)
        for (int j = 0; j < m2.m; j++)
        {
            res.matr[i][j] = 0;
            for (int k = 0; k < m1.m; k ++)
                res.matr[i][j] += m1.matr[i][k] * m2.matr[k][j];
        }
    *t = tick() - *t;
    return res;
}
```

Однопоточное умножение матриц методом Винограда

```
matrix_t vinograd_mult(const matrix_t m1, const matrix_t m2, unsigned long
long int *t)
{
    res = create_matrix(m1.n, m2.m);
    int *mul1 = malloc(sizeof(int) * m1.n ),
        *mul2 = malloc(sizeof(int) * m2.m);

    *t = tick();
    for (int i = 0; i < m1.n; i++)
    {
        mul1[i] = m1.matr[i][0] * m1.matr[i][1];
        for (int j = 1; j < m1.m / 2; j++)
            mul1[i] += m1.matr[i][2 * j] * m1.matr[i][2 * j + 1];
    }
    for (int j = 0; j < m2.m; j++)
    {
        mul2[j] = m2.matr[0][j] * m2.matr[1][j];
        for (int i = 1; i < m1.n / 2; i++)
            mul2[j] += m2.matr[2 * i][j] * m2.matr[2 * i + 1][j];
    }

    for (int i = 0; i < m1.n; i++)
        for (int j = 0; j < m2.m; j++)
        {
            res.matr[i][j] = -mul1[i] - mul2[j];
            for (int k = 0; k < m1.m / 2; k ++)
                res.matr[i][j] += (m1.matr[i][2*k+1] + m2.matr[2*k][j]) *
                                  (m1.matr[i][2*k] + m2.matr[2*k+1][j]);
        }

    if (m1.m  % 2 == 1)
        for (int i = 0; i < m1.n; i++)
            for (int j = 0; j < m2.m; j++)
                res.matr[i][j] += m1.matr[i][m1.m - 1]* m2.matr[m2.n - 1][j];
    *t = tick() - *t;

    free(mul1);
    free(mul2);
    return res;
}
```

Многопоточное умножение матриц

В каждом потоке заполняются $\frac{n}{N}$ строк результирующей матрицы, где n – размерность матрицы, а N – количество используемых потоков.

```
static void *mullthread(void *args)
{
    mullargs1 *arg = (mullargs1 *)args;
    for (int i = arg->start_row; i < arg->end_row; i++)
        for (int j = 0; j < arg->m2.m; j++)
        {
            arg->res.matr[i][j] = 0;
            for (int k = 0; k < arg->m1.m; k ++)
                arg->res.matr[i][j] += arg->m1.matr[i][k] * arg-
>m2.matr[k][j];
        }
}

matrix_t matr_mult_par(const matrix_t m1, const matrix_t m2, unsigned long
long int *t, int num_threads)
{
    matrix_t res = create_matrix(m1.n, m2.m);

    *t = tick();
    pthread_t threads[num_threads];
    mullargs1 args[num_threads];
    int row_per_thread = res.m / num_threads;
    for (int i = 0; i < num_threads; ++i)
    {
        args[i].m1 = m1;
        args[i].m2 = m2;
        args[i].res = res;
        args[i].start_row = i * row_per_thread;
        args[i].end_row = (i == num_threads - 1) ? res.n : (i + 1) *
row_per_thread;
    }

    for (int i = 0; i < num_threads; i++)
        pthread_create(&threads[i], NULL, mullthread, &args[i]);

    for (int i = 0; i < num_threads; i++)
        pthread_join(threads[i], NULL);
    *t = tick() - *t;
    return res;
}
```

Многопоточное умножение матриц методом Винограда

В каждом потоке заполняются $\frac{n}{N}$ строк результирующей матрицы, где n – размерность матрицы, а N – количество используемых потоков.
При этом векторы mull1 и mull2 заполняются до распараллеливания и передаются в конструктор для каждого из потоков.

```
static void *vinograd_thread(void *args)
{
    mullargs2 *arg = (mullargs2 *)args;
    for (int i = arg->start_row; i < arg->end_row; i++)
        for (int j = 0; j < arg->m2.m; j++)
        {
            arg->res.matr[i][j] = -arg->mul1[i] - arg->mul2[j];
```

```
            for (int k = 0; k < arg->m1.m / 2; k ++)
                arg->res.matr[i][j] += (arg->m1.matr[i][2*k+1]
                       + arg->m2.matr[2*k][j]) *(arg->m1.matr[i][2*k]
                       + arg->m2.matr[2*k+1][j]);
        }
    if (arg->m1.m  % 2 == 1)
        for (int i = arg->start_row; i < arg->m1.n; i++)
            for (int j = 0; j < arg->m2.m; j++)
                arg->res.matr[i][j] += arg->m1.matr[i][arg->m1.m - 1]
                       * arg->m2.matr[arg->m2.n - 1][j];
}
matrix_t vinograd_mult_par(const matrix_t m1, const matrix_t m2, unsigned
long long int *t, int num_threads)
{
    matrix_t res = create_matrix(m1.n, m2.m);

    int *mul1 = malloc(sizeof(int) * m1.n ),
        *mul2 = malloc(sizeof(int) * m2.m);

    *t = tick();
    for (int i = 0; i < m1.n; i++)
    {
        mul1[i] = m1.matr[i][0] * m1.matr[i][1];
        for (int j = 1; j < m1.m / 2; j++)
            mul1[i] += m1.matr[i][2 * j] * m1.matr[i][2 * j + 1];
    }

    for (int j = 0; j < m2.m; j++)
    {
        mul2[j] = m2.matr[0][j] * m2.matr[1][j];
        for (int i = 1; i < m1.n / 2; i++)
            mul2[j] += m2.matr[2 * i][j] * m2.matr[2 * i + 1][j];
    }

    pthread_t threads[num_threads];
    mullargs2 args[num_threads];
    int row_per_thread = res.m / num_threads;
    for (int i = 0; i < num_threads; ++i)
    {
        args[i].m1 = m1;
        args[i].m2 = m2;
        args[i].mul1 = mul1;
        args[i].mul2 = mul2;
        args[i].res = res;
        args[i].start_row = i * row_per_thread;
        args[i].end_row = (i == num_threads - 1) ? res.n : (i + 1) *
row_per_thread;
    }

    for (int i = 0; i < num_threads; i++)
    {
        pthread_create(&threads[i], NULL, vinograd_thread, &args[i]);
    }

    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
    *t = tick() - *t;

    free(mul1);
    free(mul2);
    return res;
}
```

Эксперимент

Были проведены замеры работы алгоримов умножения размерностями от 100*100 до 1000*1000 с шагом 100 с использованием 4 потоков , а также сравнение эффективности при разном кол-ве потоков.

*По оси ординат используется время работы в тиках *1e-6*
По оси абсцисс – размерность матрицы

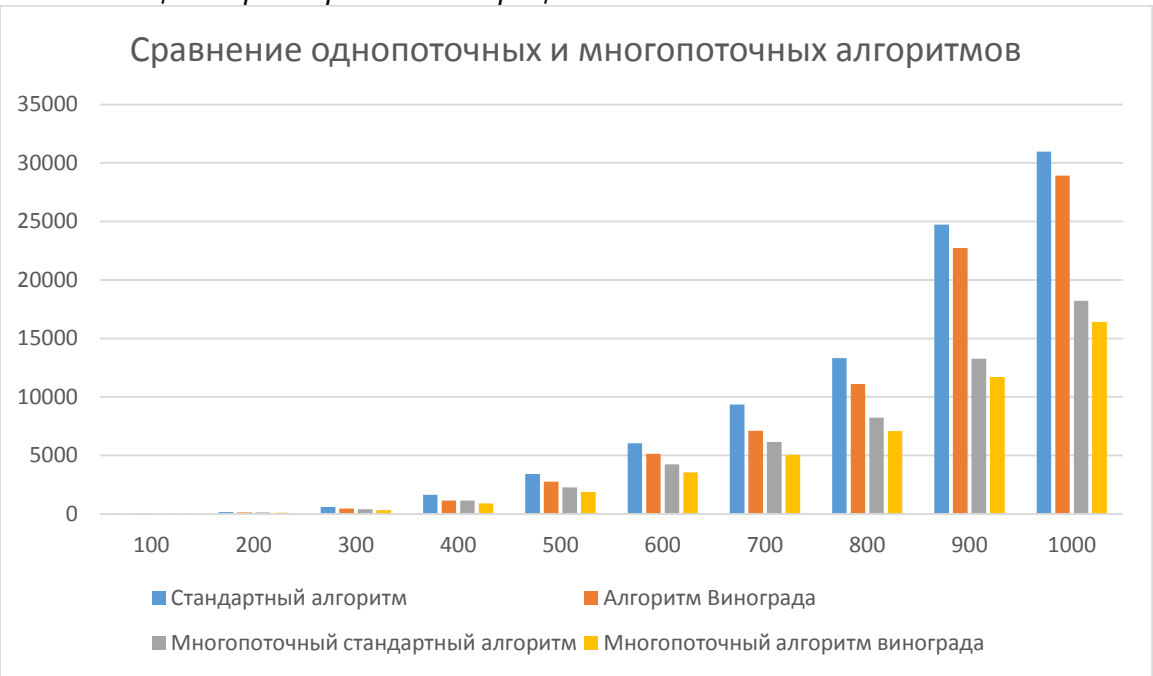

Рисунок 1 Гистограмма сравнения однопоточных и многопоточных алгоритмов

*По оси ординат используется время работы в тиках *1e-6*

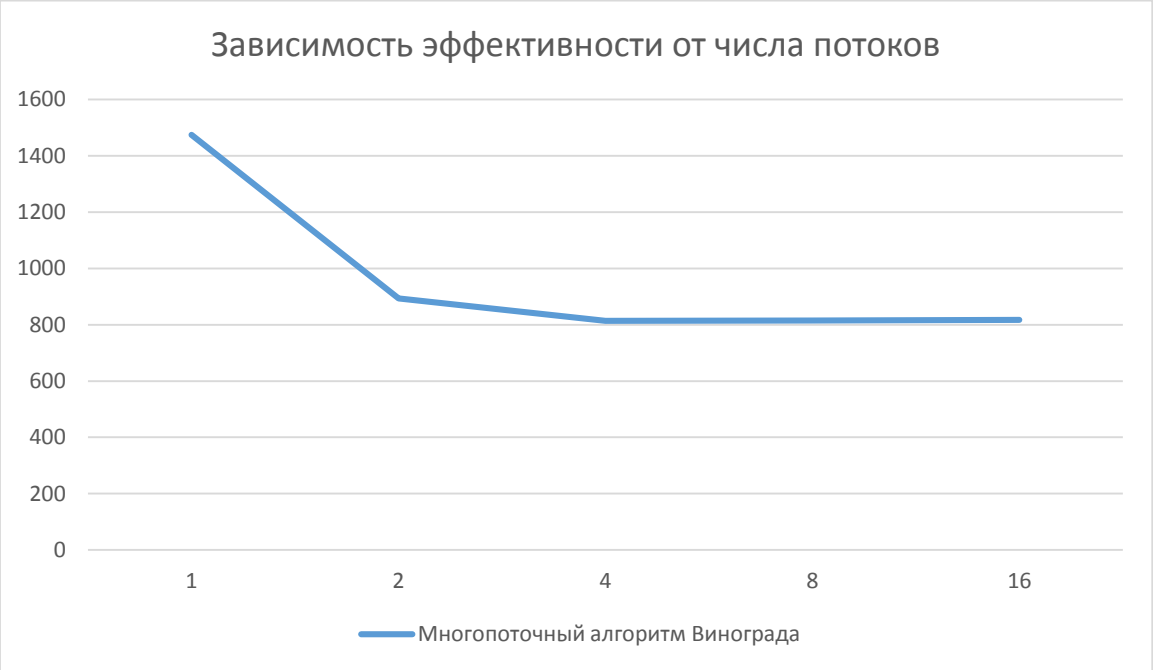

Рисунок 2 График зависимости эффективности от числа потоков

Вывод : эксперимент показал, что распараллеливание алгоритма на несколько потоков положительно сказывается на его временной эффективности. Увеличение числа потоков улучшает временные показатели до тех пор, пока число потоков не превышаетс кол-во логических ядер процессора

Заключение

Мною были реализованы алгоритмы умножения матриц средствами нескольких потоков, проведено сравнение однопоточных алгоритмов с многопоточными. Экспериментальные результаты показали эффективность многопоточного умножения матрицю